REAL-TIME COMPUTER CONTROL OF A PROTOTYPE BIPEDAL SYSTEM

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Bachelor of Science with Distinction

at The Ohio State University

By

Patrick Wensing

* * * * *

The Ohio State University

2009

Honors Examination Committee:

Dr. David E. Orin, Adviser

Dr. Yuan F. Zheng

Approved by

Adviser
Electrical and Computer Engineering

ABSTRACT

While much work in the area of robotics has attempted to replicate the power and agility of biological locomotion in a physical system, even the most impressive prototypes to date are functionally very simple in comparison to their biological counterparts. Biological systems particularly excel in their performance of dynamic maneuvers, such as a run or a jump. These maneuvers require explosive leg power coupled with sudden changes in trajectory. In mechanical systems, these requirements create the need for high-performance actuation and fast real-time control. Through a fusion of biologically inspired design and intelligent control strategies, current work aims to perform these types of maneuvers on a prototype biped robot named KURMET. As part of this research thrust, the goal of this work was to establish real-time control for the bipedal system. A distributed control system, that uses a cutting edge motion controller in conjunction with a Linux host computer, was developed to control these maneuvers. The resulting system provides on-board real-time control capable of 1 kHz servo rates over the biped's four actuators. A distributed control framework was developed to interface this foundational control system with the Linux host. This framework was then applied to produce state-based control strategies which have demonstrated a walk and a high-performance jump in hardware. While applied to these two complex movements, the control framework's modular design facilitates

extension to a wide range of motions. Thus, this work has laid the foundation for a rich set of investigations into dynamic maneuvers for this platform.

To my family

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. David Orin, for providing the opportunity to work on this multi-disciplinary project. His availability, understanding, and insights throughout the project have caused this past year to be a rewarding learning experience. Additionally, I would like to thank Dr. Jim Schmeideler for his advice throughout the project. His welcoming demeanor and encouraging comments have always been appreciated, especially upon first joining the group. Finally, I'd like to thank Dr. Yuan F. Zheng for taking the time to serve on my committee.

I would not have been able to complete this work without the help of fellow group members Yiping Liu and Ghassan Bin Hamman. Yiping, you are a master of the soldering iron and have helped me to understand the robot through our many discussions. Ghassan, you have been a great resource throughout the project. Whether with software, circuits, or the presentation of my work, your comments have been very keen, and always constructive. On the mechanical side, I'm grateful to Brian Knox for designing such a meticulously engineered system. I would also like to express my thanks to Matt Hester for all his assistance with experimentation. His patience at explaining anything mechanical has been greatly appreciated throughout the project.

Finally, I'd like to thank my family for all their support these past few years. Their constant encouragement and advice throughout my academic career have been, in every way, a blessing.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

Introduction

Although much research has attempted to harness the power and efficiency of human locomotion in mechanical systems, a solution to this problem is still in its infancy. Human legs and the associated neurocontrol system provide extreme agility in a variety of terrain conditions that is unparalleled by any physical system to date. Much of this agility can be attributed to biological systems' ability to execute truly dynamic movements. These movements are characterized by the maintenance of stability through sudden changes in speed or trajectory with significant acceleration [4]. In mechanical systems, performance of these movements creates the need for high-performance actuation and fast real-time control. These movements often involve significant periods of flight, during which the system is largely uncontrollable. Thus, control strategies must be devised which plan a feasible trajectory through the uncontrolled portion of state space with the knowledge that control will be regained at the other side [5].

Through a fusion of biologically inspired design and intelligent control strategies, current work aims to perform these types of maneuvers on a prototype biped robot, shown in Fig. 1.1. The prototype is named KURMET, which is an acronym for "**K**inematically **U**nderactuated **R**obot for dynamic **M**aneuver **E**xperimental **T**esting".

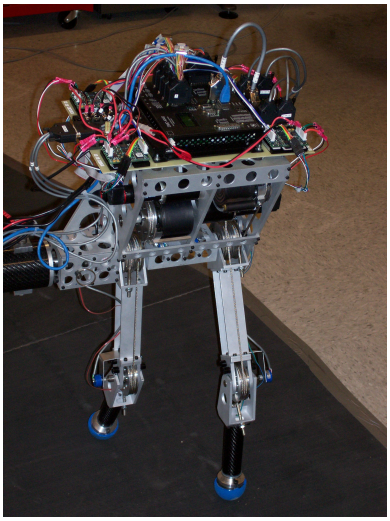

Figure 1.1: KURMET: A prototype biped for the study of dynamic movements

1.1 Motivation

Researchers have long sought to develop legged machines which capture the advantages that legs offer, in terms of mobility and agility, in the biological world. Unlike wheeled or tracked vehicles, that require a continuous path, legged vehicles can navigate highly irregular terrain using only discrete footholds for traction and support [6]. This provides legged platforms with an advantage for applications to replace humans in a variety of hazardous occupations. Potential applications include a spectrum from search-and-rescue operations, to military surveillance, to scientific exploration in harsh environments.

Work on a bipedal system is particularly motivated by applications to assistive robotics. The bipedal structure is a natural choice for this application since human environments are designed for bipedal accessibility. Still, for a bipedal machine to

operate in a realistic environment, it must be able to perform more than just a statically stable gait. These robots must be endowed with the ability to perform dynamic movements in order to quickly and robustly adapt to their surroundings.

The insights gained into a biped's performance of dynamic movements can contribute to the area of rehabilitation robotics as well. Gait-training robots have been in use since the 1990's to help stroke patients regain their ability to walk. These machines support a portion of the weight of a patient while their legs are actively moved through a gait. Current efforts are underway to let the patients contribute to their own locomotion as much as possible with motion assistance only as needed [7]. This has required identification of the most important aspects of assistance. Yet, these efforts have focused on quasi-static gaits, lacking the ability to target true dynamic movement effectively. A more thorough understanding of dynamic movement will allow this type of therapy to rehabilitate a wider range of movement. In addition, the design of robotic systems for execution of these movements can inform the design of robotic prosthesis.

1.2 Previous Research

A number of investigations into legged robotics at Ohio State have motivated the construction and control of KURMET for the performance of dynamic movements. The single-leg hopper, built by Remic [8], was developed to investigate the use of a lightweight prototype leg with series-elastic actuation in a quadrupedal machine capable of a planar gallop. The prototype is a single articulated leg, with two revolute joints at the hip and knee, constrained to vertical motion by four rails. The hip employs a direct drive actuator, while the knee uses a biologically inspired series-elastic

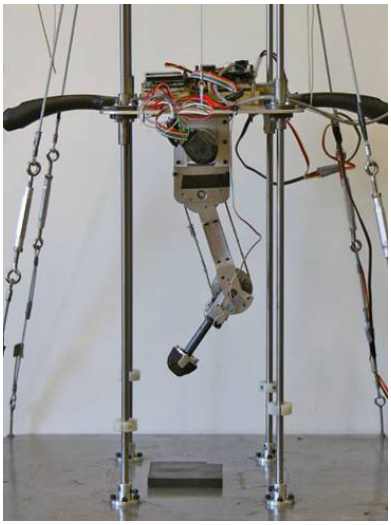

Figure 1.2: The Hopper: A series-elastic actuated, articulated leg

actuator (SEA). The SEA was later modified by Knox [9] to include a unidirectional aspect which improved the leg's positioning capabilities during flight. The initial control hardware and software was developed by Curran [10] and demonstrated a high jump in hardware. The control hardware was modified by Huang [11] and Birkmeyer [12] to perform repetitive jumping and real-time leg cycling. Later work by Curran investigated the optimality of the SEA design through the use of an evolutionary search algorithm [13]. This work demonstrated the desirability of series-elastic actuation at both the hip and knee, as opposed to simply the knee as in the Hopper. The combination of these efforts has motivated the inclusion of articulated legs with SEAs into a less constrained legged platform for the execution of dynamic movement.

Series-elastic actuators have been used successfully in a number of other dynamic robots. They have been used effectively for dynamic walking as demonstrated by Pratt et al. in Spring Flamingo [14] and by Grizzle and Hurst in MABLE [15]. While

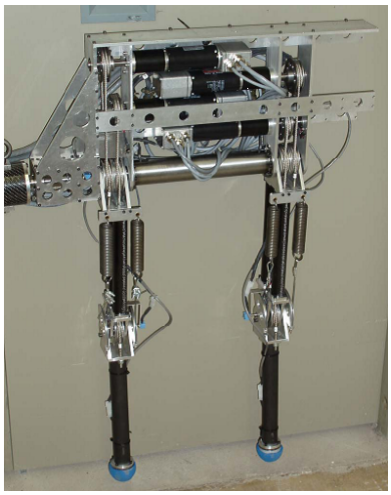

Figure 1.3: ERNIE: a planar biped for the study of parallel knee compliance in dynamic walking

MABLE has demonstrated dynamic walking, its ultimate goal is the performance of dynamic running. A jumping robot, Mowgli, has used SEAs to produce jump heights of half its body height [16].

Recent design optimizations by Knox [17] have produced KURMET, a biped for the study of dynamic movements. KURMET's design has been heavily motivated by the Hopper and by ERNIE. ERNIE, developed at Ohio State, is a planar biped for the study of parallel knee-joint compliance in dynamic walking [18]. A photograph of ERNIE is shown in Fig. 1.3. Work by Knox on KURMET has optimized the system's structural components, series-elastic actuators, motors, and motor drive system. While the design was specifically optimized for performance of dynamic jumping and running, these maneuvers provide a foundation for the execution of more complex dynamic maneuvers. As a result, the system should be capable of delivering the large accelerations needed for a wide range of dynamic motion.

Previous investigations into the use of intelligent control have informed its application to the control of dynamic maneuvers for KURMET. Successful execution of the dynamic maneuvers requires the coordination of a high degree-of-freedom system, maintenance of its stability, and control of motions with significant periods of flight during which the system is largely uncontrollable. As a result of these complexities, in general the closed-form mathematical equations which govern the dynamic maneuver cannot be derived, which prevents the application of many standard control techniques. Intelligent control strategies provide a tractable approach to address these complexities. Curran used an evolutionary search algorithm to find control parameters for a maximal-height jump [13]. Krasny and Orin employed a multi-objective genetic algorithm with a modular controller to find solutions for complex dynamic movements in a 3D quadruped [4]. Palmer and Orin applied intelligent control to perform a quadruped trot while turning in simulation [19].

1.3 Objectives

The high-level objective of this research is to more fully understand the notions of dynamic stability and to advance the performance of dynamic maneuvers in biped robots. Due to the highly open nature of this problem, the more narrow goal of the research's current effort is the performance of dynamic maneuvers for the experimental biped KURMET. This effort is cross-disciplinary, with design and control investigation occurring collaboratively across mechanical and electrical engineering. This thesis focuses specifically on the establishment of distributed real-time control for KURMET.

The first objective of this work is to develop a real-time distributed control system for the prototype bipedal system. Responsive real-time control is essential for the performance of dynamic maneuvers. For the prototype system, this involves control that is distributed across an on-board processor and an off-board host processor. It is desired for the designed structure to be modular so that it may be applied broadly for all movements on the platform.

The second objective is to develop the low-level control software for the execution of repetitive jumping. Concurrent investigations into control for the biped aim to develop an intelligent controller for the supervisory control of repetitive, precise-height jumping. These efforts require low-level control software to perform continuous control of the jump while the supervisory control dictates the high-level control policy at discrete intervals (once per jump).

The final objective is to design and implement a controller for the performance of a locked-torso walk. This involves the synthesis and verification of a high-level speed controller and motion planner. In addition, it involves integration of the strategy with the distributed control framework for verification of the control approach in hardware.

1.4 Organization

The remainder of this thesis will be organized as follows. The first two chapters will outline the control details for the platform that are common to all types of movements. Chapter 2 will present an overview of the mechanical components of the system and will give an in depth look at the foundational control system. This will include a description of the biped's sensory components, electrical interface to the actuators, and an explanation of the low-level controls. Chapter 3 will examine the

distributed control framework, developed to facilitate layered control approaches for the biped.

The next two chapters will provide details of the application of this general framework to the execution of two specific movements. Chapter 4 will present the implementation of a real-time jump controller. While much of the control strategy was devised by work in [20], this chapter highlights hardware implementation details. Chapter 5 will detail the development and implementation of a novel control strategy for performance of a locked-torso walk. Finally, Chapter 6 will present conclusions for the work and outline potential research topics for the biped.

CHAPTER 2

Foundational Control System: Actuation, Sensing, and Control

2.1 Introduction

This chapter will outline the foundational control system and hardware for real-time control of the biped's movement. The execution of high-speed dynamic maneuvers requires explosive leg power coupled with sudden changes in trajectory [13]. In physical systems, these requirements create the need for high-performance actuation and fast real-time control. In contradiction to these requirements, execution of these maneuvers requires the physical system to be light-weight. In order to minimize weight, while still providing the benefit of explosive leg power, the bipedal system employs series-elastic actuation at its joints. The block diagram for a series-elastic actuator (SEA) is shown in Fig. 2.1. The electro-mechanical interface for the each SEA is provided by a high-performance 3-phase brushless direct current (DC) motor. The motors are driven by four compact motor amplifiers, each capable of delivering 12A of peak current.

Additionally, the high speed of dynamic maneuvers requires control decisions to be made on a millisecond time scale. Palmer and Orin [19] have noted that running

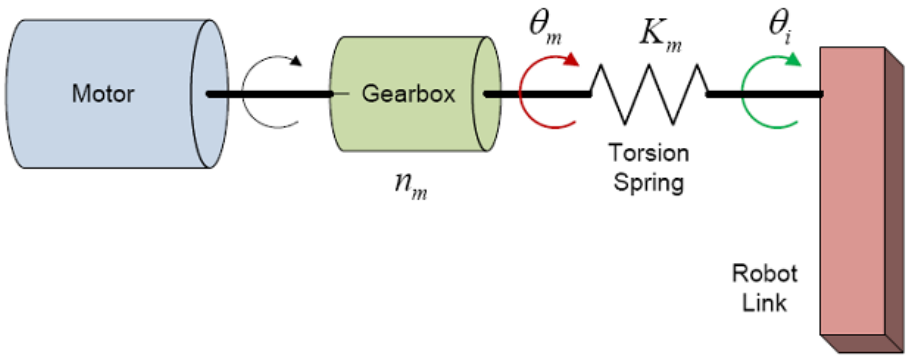

Figure 2.1: SEA block diagram from [17]

strides involve significant periods of flight during which the body is largely uncontrollable. Corrective forces must be applied during short stance intervals when the compliant elements and high-power actuators work in combination to provide considerable leg power. As a result, it is critical that sensing and control happen in a millisecond time period to provide the necessary corrective measures during these short stance intervals. In order to provide this level of responsivity, a state-of-the-art motion controller was chosen to perform real-time control of the biped's four actuated degrees of freedom. The motion controller uses a 100Mbs Ethernet connection with a Linux host to enable low-latency changes to its control strategies.

The foundational control system, present on the motion controller, makes use of a network of sensors to deduce the kinematic state of the biped. Encoders are used to measure the position of each motor as well as the position and orientation of the torso. In addition to the motor positions, the orientation of each link is measured through the use of a potentiometer. Finally, ground contact is sensed by digital switches

placed at the base of each foot. The high-bandwidth feedback from this network of sensors is necessary for responsive real-time control of the system.

2.2 System Overview

Prior to a detailed examination of the foundational control system, the next two sections will provide a brief overview of the mechanical and electrical systems for the biped. Figure 2.2 shows a high-level overview of the mechanical system and its interface with the biped's electronics. Repeated details for the left and right legs are omitted for clarity.

2.2.1 Mechanical System Overview

The mechanical system is mainly comprised of a boom for stabilization, the SEAs found in the torso, and a pair of articulated legs. Each leg has two revloute joints actuated in parallel with respect to the body. That is, when the knee motor is held in position and the thigh motor is displaced, the shank will not change orientation relative to the body. This is in contrast to serial actuation, where, under a similar situation, the shank would not change orientation with respect to the thigh. As shown in Fig. 2.2, each link employs a unidirectional series-elastic actuator (USEA). This design allows direct drive between the motor and link in one direction, and adds an element of compliance between the motor and link in the other. To this effect, the motor is directly coupled to either a spiral torsion spring or a stiff contact, which then transfers power to the link. The compliant element stores energy during stance phases of dynamic maneuvers, which allows for higher lift-off velocities. In flight, the unidirectional aspect has been shown to improve foot positioning [13]. Additionally,

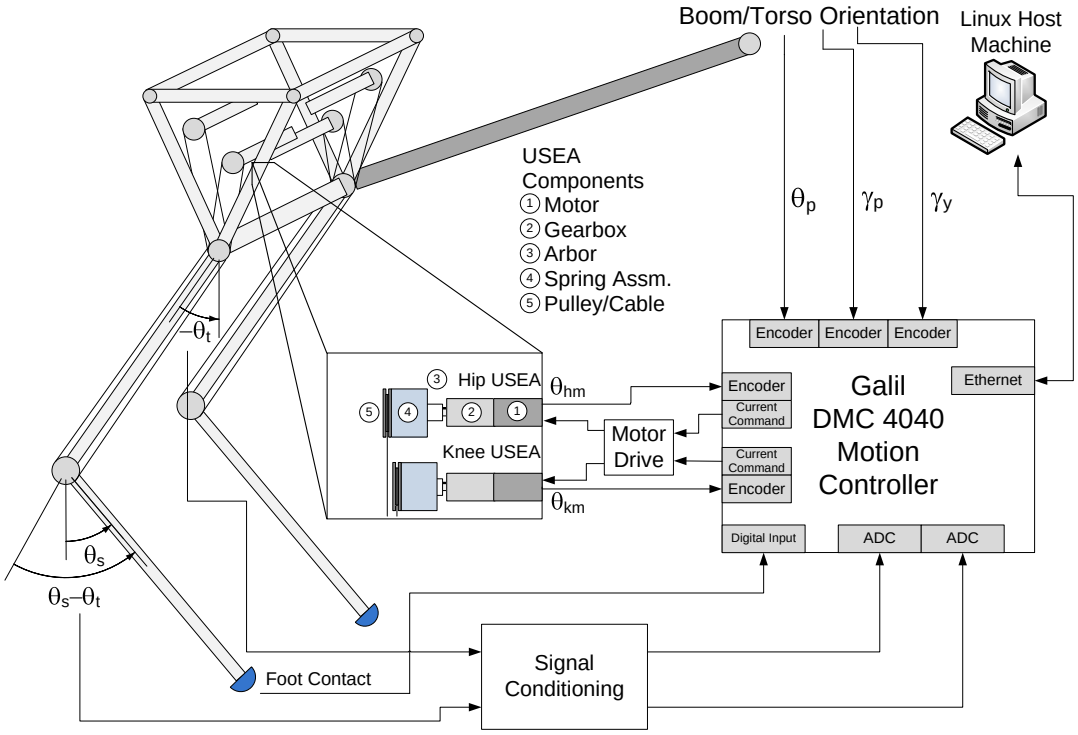

Figure 2.2: Mechanical and electrical system interface

A pre-load is placed on the torsional spring to prevent spring deflection when high torques are not required, such as during flight.

To provide direct physical intuition, with compliance engaged, the system can be viewed as shown in Fig. 2.3. The following notation is introduced and will be used throughout the rest of the document. The coordinate system will be rigidly attached to the biped at the outer edge of its right hip. Let the joint angles, as measured at the output of the gearbox, be θ_h and θ_k for the hip and knee motors respectively.

Angles measured at the input of the gearbox are denoted θ_{hm} and θ_{km}. Also, let the thigh and shank link angles be θ_t and θ_s with torques τ_t and τ_s correspondingly.

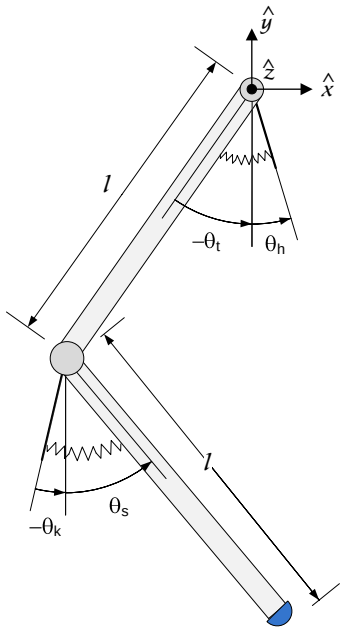

Figure 2.3: Kinematic model of a single leg. Note: All angles taken from the vertical.

It should be noted that, at the hip, the compliance may become engaged when the motor moves in the positive θ_h direction, or when the link moves in the negative θ_t direction. The opposite holds true for the knee. Letting k be the spring constant of the spiral torsion springs, the link torques can be found as a function of the motor and link positions. From this point on, the vector notation will be dropped with the understanding that all torques will be taken about the $+\hat{z}$ axis.

$$\vec{\tau_t} = k(\theta_t - \theta_h)\hat{z} \tag{2.1}$$

$$\vec{\tau_s} = k(\theta_s - \theta_k)\hat{z} \tag{2.2}$$

Instead of being free to move in 3-space, the biped is stabilized by a boom as shown in Fig. 2.4, which approximates a planar restriction of motion. The boom is attached to the biped at an unactuated revolute joint located at the outer edge of the right hip. This degree of freedom allows the biped to pitch forward and backward in its sagittal plane. Let the boom's pitch and yaw be defined as γ_p and γ_y. Additionally, let the measure of torso pitch be defined as θ_p. For all work discussed in this document, a pin has been used in hardware to prevent torso pitch.

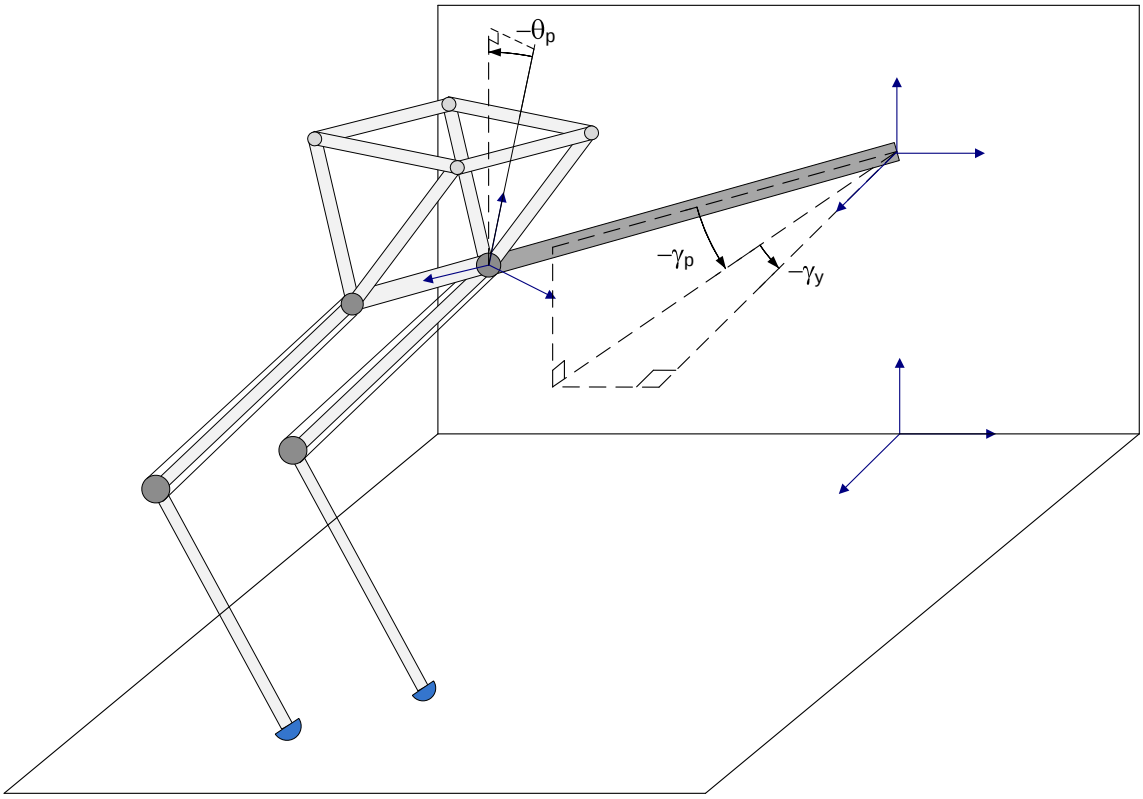

Figure 2.4: Description of boom and torso angles

Numeric values for the biped's physical parameters can be found in Section A.1. In addition, a more comprehensive description of this system and its design can be found in [17].

2.2.2 Electrical System Overview

The electrical system is comprised of a variety of sub-systems including the actuation, sensors, power, and control. Other than the power system and boom encoders, all electrical hardware for the foundational control system is present on board the biped. Figure 2.5 shows a connection diagram for the electrical system. As shown on this figure, both 110VAC and 220VAC are used to power the bipedal system. The off-board drive power system uses 220VAC and directly provides power to the motor drives. An additional 110VAC power supply is used to power the motion controller and sensory electronics.

At the center of the electrical system is a state-of-the-art motional controller, the Galil DMC-4040. The motion controller closes the position loop around the motor and amplifier through feedback from an incremental encoder. The amplifier then closes the current loop around the motor through current feedback. These two feedback loops constitute the extent of the control performed by the foundational control system.

In addition to the sensory information needed for positional feedback, the motion controller gathers all other sensory information needed for high-level control. A set of boom encoders interface with the auxiliary encoder connections on the motion controller. The Galil also provides digital and analog input to gather foot contact and link position data respectively. Link position data is inferred through the analog voltages from potentiometers. However, these potentiometer voltages are conditioned

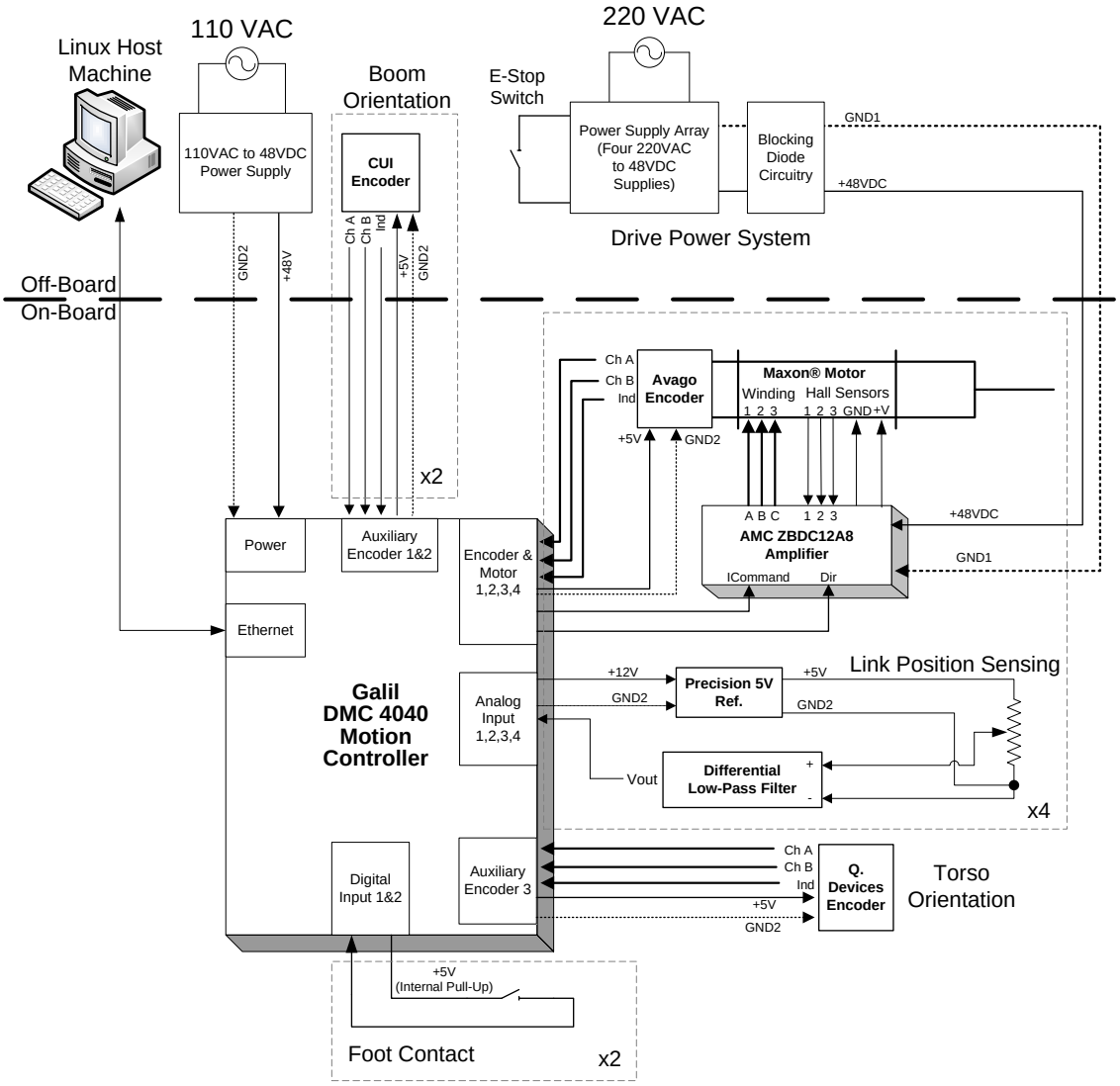

Figure 2.5: Overall electrical system diagram

by custom circuitry prior to being interfaced with the Galil. All of the gathered sensory data is relayed to a host machine over Ethernet for use in complex calculations and high-level motion planning.

2.3 Motor Interface

The interface from the electrical system of the biped to its mechanical system is provided through four 3-phase brushless DC motors. Through optimization of jump height performed in [17] the Maxon Powermax EC-30 was selected for use in KURMET's USEAs. A gear ratio of 126:1 was selected for the motor's gearbox. These high-performance actuators operate nominally at 48V with a maximum continuous current of 4.7A [21]. The motors are electrically interfaced to the amplifier to receive stator winding currents, and to the motion controller for position feedback. The next two sections will discuss the specifics of these connections as shown in Fig. 2.6.

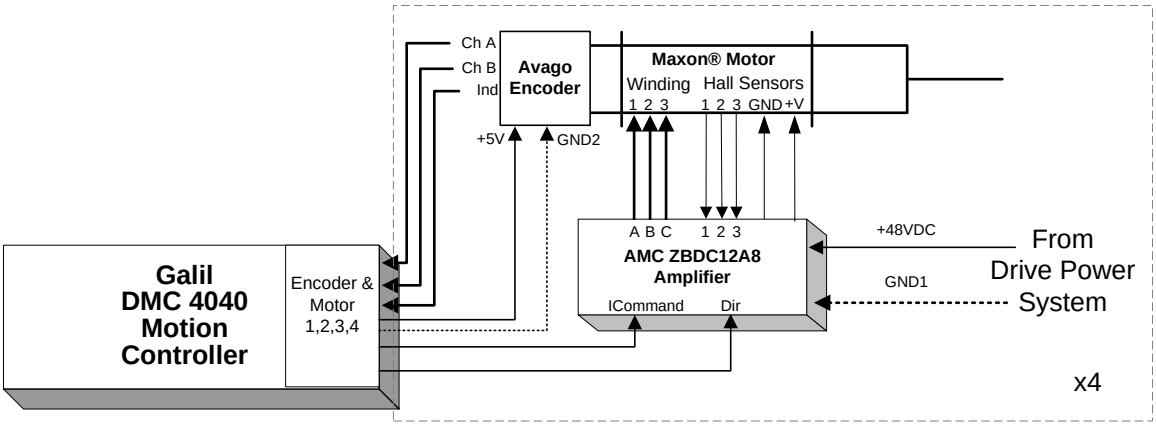

Figure 2.6: Motor, drive, and encoder interface diagram

2.3.1 Amplifier Setup

Four amplifiers provide a compact, light-weight, drive system for the Maxon DC motors. Work in [17] found the Advance Motion Controls ZBDC12A8 to be an optimal choice for KURMET's drive system. The ZBDC12A8 is a torque-mode amplifier which performs closed loop current control and hall sensor commutation. The amplifier uses a PWM and direction input to qualify the desired current. A peak current of 12A can be attained for a maximum of 2 seconds, while 6A can be supplied continuously [22].

Torque in the brushless DC motors is supplied by the magnetic field interaction of the permanent magnet rotor and the induced magnetic field in the 3-phase stator windings. Optimal torque per ampere of current is achieved when these two fields are orthogonal. To this end, the magnetic field from the stator windings must be controlled to stay nearly perpendicular to the rotor magnet. In the amplifier, feedback from the motor's three hall effect sensors is decoded to infer the position of the rotor. This position determines which pair of winding phases will be driven by a MOSFET drive. As a result, current in the stator windings produces a magnetic field at an angle that is optimal within $\pm 30^\circ$ and a torque to current ratio that is optimal within 13% [23]. The correct connection of feedback from the hall effect sensors is paramount to achieve this torque performance per amp of current.

In addition, torque control requires current control. Current control is provided by the ZBDC12A8 in the form of current feedback and an analog PI current loop. The proportional and integral gains for the amplifier were chosen by the manufacturer to provide adequate performance in most motor control applications. These gains were found to be acceptable and were not modified.

As shown, the amplifier has many additional connections. The amplifiers receive +48V from the drive power system, and deliver this power to the amplifiers via 3 motor-phase connections. Power to the motor phases is switched at a frequency of 33kHz. As mentioned previously, the Galil interfaces to each amplifier through PWM and direction digital signals to provide a current command. The duty cycle of this PWM signal represents the percentage of max current (12A) that is desired. A more detailed explanation of the amplifier wiring can be found in Sections D.3.1 and D.3.1.

2.3.2 Encoder Setup

Each motor has an incremental encoder that is used to close its position loop on the Galil. The motor encoder used is the Avago HEDL5540. It is a 500 line quadrature encoder with differential outputs [3, 24]. Since the encoder is placed on the input side of the motor gearbox, the resulting resolution is 252000 quadrature counts per rotation of the gearbox output. Each encoder is interfaced directly to the Galil's quadrature encoder input for its corresponding axis.

Noise issues were found to be negligible for the motor encoders. These encoders, located at the end of each motor, are within a foot of their connections on the motion controller. The differential signaling for the encoder signals allows the differential receiver in the motion controller to reject common mode noise between each signals' pair of lines. This makes the transmission much more immune to noise issues. This result, combined with the short length of the cable informs the decision to neglect shielding. For more details on the exact connections refer to Section D.3.2.

2.3.3 Motor Encoder Calibration

The motor encoders are calibrated through a two step calibration sequence for each leg. Let us denote the angles from the uncalibrated encoders θ'_h and θ'_k for the hip and knee respectively. The calibration sequence makes use of physical joint hard stops that limit the hip and knee motion. These hard stops are located at the joint between the torso and thigh, and at the joint between the thigh and shank. Thus, the shank hard stops limit its motion relative to the thigh, yet it is actuated relative to the torso. In the first step of the calibration sequence, the hip actuator is left unpowered while the knee actuator is driven in the negative θ_k direction. Once the knee hard stop is engaged, the shank is no longer able to rotate relative to the thigh. Yet, due to the parallel actuation, any further knee motor displacement will cause the knee to change orientation relative to the torso. As a result, the unactuated thigh begins to deflect until reaching its hard stop. Once both hard stops are engaged, the encoder positions are recorded as $\theta'_{h_{min}}$ and $\theta'_{k_{min}}$. In the second step, the knee motor is driven in the opposite direction until the opposite hard stops are engaged. Once again the encoder positions are recorded as $\theta'_{h_{max}}$ and $\theta'_{k_{max}}$.

The zero-configuration for the motors, with the leg stretched to singularity directly beneath the hip, is then estimated based on the extremes of the motor positions recorded. First, due to hip hard stops symmetric location with respect to $\theta_t = 0$, the hip zero position is found as follows:

$$\theta'_{h_{zero}} = \frac{\theta'_{h_{max}} + \theta'_{h_{min}}}{2} \tag{2.3}$$

Due to the shanks motion being limited relative to the thigh, but actuated relative to the torso, its zero configuration motor location will be calculated in three steps.

First, if the hip starts from the configuration against the minimum joint stops and moves to the zero configuration, the hip has moved by:

$$\Delta\theta'_h = \theta'_{h_{zero}} - \theta'_{h_{min}} \tag{2.4}$$

If the knee is to maintain its orientation relative to the thigh, at the minimum joint stop, it will have to deflect equally. Second, with the hip locked at its zero position, the possible range range of knee angles will be of size:

$$\theta'_{k_{range}} = (\theta'_{k_{max}} - \theta'_{k_{min}}) - (\theta'_{h_{max}} - \theta'_{h_{min}}) \tag{2.5}$$

Now, based on the KURMET's design, there is a theoretical range for the shank denoted $\theta_{s_{range}}$. Let the measured error of this range be defined as:

$$e'_{k_{range}} = \theta_{s_{range}} - \theta'_{k_{range}} \tag{2.6}$$

Finally, let $\theta_{s_{min}}$ be the theoretical minimum shank angle for the hip at singularity. The knee zero is then defined as:

$$\theta_{k_{zero}} = \theta_{k_{min}} + \Delta\theta'_h - \theta_{s_{min}} + \rho * e'_{k_{range}} \tag{2.7}$$

Here ρ is ideally .5 if the error in achievable joint limits is distributed evenly forward and backwards. It was found experimentally to be .65.

2.4 Boom Encoders

Two incremental boom encoders are used to determine the location of the boom with respect to the inertial reference frame. The boom encoder used is the CUI Inc. NSO-S10000-2MD-8-050. It is a 10000 line quadrature encoder with differential outputs [25]. After quadrature, it provides 40000 counts per revolution. This number

does not present a clear picture of the encoders resolution over the workable range of the biped, as experiments often will only utilize 5% of the encoders full range. The boom yaw is connected to the Galil auxiliary encoder input A, while the boom pitch is connected to the auxiliary encoder input B.

In contrast to the motor encoders, noise issues were fully addressed with the wiring of the boom encoders. The boom encoder signals travel 2 meters across the boom, in close proximity to the system power lines. As a result, each differential signal was transmitted through a shielded twisted pair cable and was kept as far away from the power lines as possible. For more details on the exact connections refer to Section D.3.3.

Additionally, a single incremental encoder is used to determine the torso pitch of the biped. This encoder is a Quantum Devices QD145-05/05-4096-0-03-T3-01-00. It is a 4096 line quadrature encoder with TTL outputs [26]. After quadrature it provides 16348 counts per revolution. Hard physical limits on the boom limit the torso pitch to a range of 65°, thus once again the the working range will not utilize all 16348 quadrature counts. Due to the short length of the cable, shielding was not used for the torso pitch encoder. Once again further information can be found in Section D.3.3.

2.5 Link Position Sensing

Due to the possibility of spring deflection in the biped's SEAs, the link angles may be different from the motor angles. Thus, to achieve the full kinematic state of the biped, link orientations must be sensed in addition to motor angles. Since the boom encoders had already occupied three of the four auxiliary encoder inputs,

another method of position sensing had to be employed. Potentiometers were selected to translate the joint positions into analog voltages. Although potentiometers are not suitable for industrial applications, the million turn rotational lives on many potentiometers is suitable for this prototype system.

2.5.1 Mechanical Setup

The potentiometer selected for link position sensing was the high-precision Bournes 6637S-1-102 single-turn potentiometer. Although the potentiometer linearity was quoted to be $\pm 1\%$ [27], it was found to be closer to $\pm .25\%$. While other types of potentiometers offered higher linearity, their increased size was not feasible for mounting purposes. Each potentiometer was mounted at the physical hip or knee joint [1]. To this effect, the thigh potentiometer measures the angle of the thigh relative to the torso, while the shank potentiometer measures the angle of the shank relative to the thigh. Thus, based on the convention developed previously, the shank angle, θ_s, must be calculated as the sum of the link angles found from the potentiometers. More detailed information on the potentiometer mounting can be found in [17].

2.5.2 Electrical Setup and Signal Conditioning Circuitry

Previous work in [12] has shown that unconditioned analog potentiometer signals can have significant noise, which makes them difficult for use in feedback. Examination of unconditioned potentiometer voltages with the Galil's on-board 12-bit ADC showed noise fluctuations in up to 4 of its least significant bits. A number of improvements were made which improved noise to 1, and occasionally 2, bits of fluctuation.

[1] As opposed to at the hip or knee SEA

First, a precision 5V reference was added to ensure a constant voltage across the potentiometer. Figure 2.7 shows the electrical setup for the link position sensing. As shown, +12V from the motion controller is passed through a resistor and shunt regulator to provide a consistent 5V reference. Due to a 15mA current restriction on the regulator, coupled with the 5mA current draw of the potentiometer, each leg was given its own precision reference. The ground and precision reference are passed as a shielded twisted pair to the outer terminals of the potentiometer. Next, the

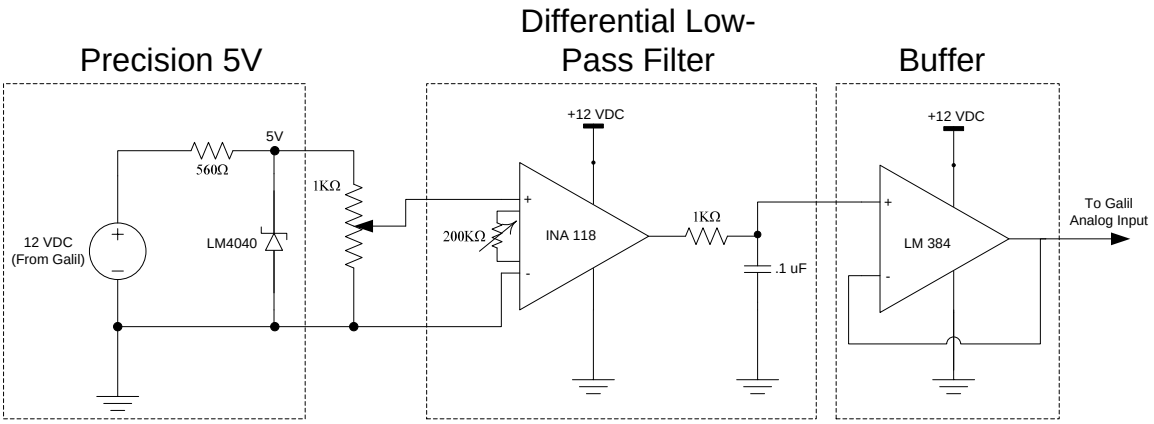

Figure 2.7: Link position sensing circuitry

ground and wiper voltage are passed via a shielded twisted pair cable to a two-stage signal conditioning circuit. In the first stage, an instrumentation amplifier removes any common mode noise picked up by the pair in transmission. A trim potentiometer is used to select the differential gain on this amplifier. Since the joints do not use all 360° of motion provided by the potentiometer, the resultant potentiometer voltage does not use the full 5V range. Thus each trim pot was adjusted so that the output

of the instrumentation amplifier would realize the full 0-5V range over the kinematic range of the link. This correspondingly maximized the position resolution that would result from the digitization of the analog signal. The output of the instrumentation amplifier is then passed through a 1st order low-pass filter to eliminate non-common mode noise. The bandwidth of this filter is set to 1.59 kHz (10,000 rad/sec) in order to substantially attenuate the 33 kHz noise from the high-power amplifier switching. The delay of filter is negligible in comparison to the maximum 500 Hz sampling rate on the motion controller.

An array of four signal conditioning circuits was fabricated onto a PCB for use on the biped. The conditioned voltages are relayed a distance of 10cm to the Galil via a shielded cable. Special noise suppression D-Sub backshells were used on both ends of the cable to further ensure clean transmission of the conditioned signals. Figure 2.8 shows an inferred link position with and without conditioning circuitry, during amplifier operation. The conditioned position values experienced 80% less noise on average than the unconditioned ones.

2.5.3 Initial Potentiometer Calibration

Due to the presence of the gain selection trim potentiometers, each link has a separate conversion factor relating its change in output voltage to a change in link position. An initial calibration was performed to determine this conversion factor for each link. For the thigh potentiometers, the knee was left unactuated while the hip was driven through its range of motion with a constant velocity. The thigh potentiometer voltage and hip motor encoder counts were recorded over this time

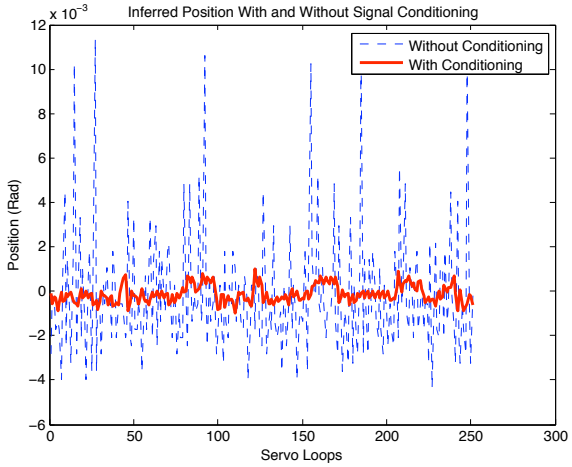

Figure 2.8: Link position sensing conditioned vs. unconditioned performance

period. Finally, the two quantities were plotted against one another and a least-squares regression line was found. The slope of this regression line, $\Delta V/\Delta\theta_h$, provided the required conversion factor. For the shank potentiometers, the hip motor was servoed to a constant position while the shank was driven through its range of motion. The same mathematical techniques were employed.

2.6 Contact Sensing

Ground contact switches are employed at the base of each foot. The design of the foot switches prevents a false trigger under inertial loads, yet allows them to be triggered under a wide range of foot impact angles. The full details of their design can be found in [17]. The foot switches use two of the motion controller's optically-isolated digital input ports. These ports are internally tied to a pull up resistor, and are shorted to ground upon closure of the switch.

2.7 System Power Configuration

Due to the large size and weight of available AC-to-DC power supplies, the bipedal system uses off-board power that is delivered across the boom. Table 2.1 details the continuous and peak power requirements for the motor drive system.

Component	**Quantity**	**Cont. (W)**	**Total Cont. (W)**	**Peak (W)**	**Total Peak(W)**
Amplifiers	4	24	96	48	192
Motors	4	288	1152	576	2304
Schottky Diode	1	17.8	17.8	35.5	35.5
Total			1265.8		2531.5

Table 2.1: Breakdown of power requirements for the motor drive system

In order to provide redundancy, a multiple power supply solution was developed. This solution incorporates four Lambda SWS600L power supplies. The supplies can operate at a continuous power of 624W and a peak power of 720W [1]. Through parallel operation of these supplies, the continuous and peak power requirements were met.

2.7.1 Power Distribution Circuitry

The Lambda power supplies offer remote on/off capabilities and parallel operation through specific wiring of their control terminals. Remote on/off capabilities were implemented to provide the ability to cut power in the case of an emergency during testing. Each power supply was wired as shown in Fig. 2.9, with the emergency switch placed in parallel for all supplies. The emergency stop switch is normally

closed, which shorts the solid state relay that is used to disable the supplies. In

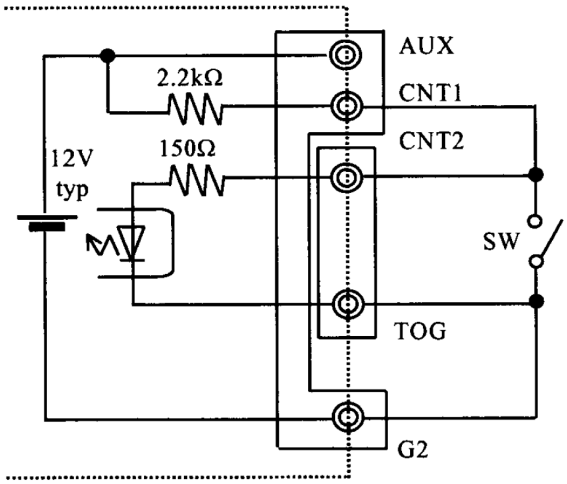

Figure 2.9: Remote on/off wiring from [1]

addition, the supply outputs were connected in parallel to allow a single pair of wires for power transmission across the boom. In this configuration, the net current draw from the motor drive system is distributed evenly between the four supplies. Figure 2.10 shows the wiring that allows this current sharing mode of operation. When extended to four supplies, all PC and all COM terminals are separately tied together. One drawback of the supplies' parallel operation is that their peak power falls by ten percent. Still, the four supplies in parallel provide a peak power of 2592W, which meets requirements.

2.7.2 Blocking Diode Circuitry

Due to the amplifiers regenerative mode of operation, when the motor is back-driven, the amplifiers can potentially provide a current back to the power supplies.

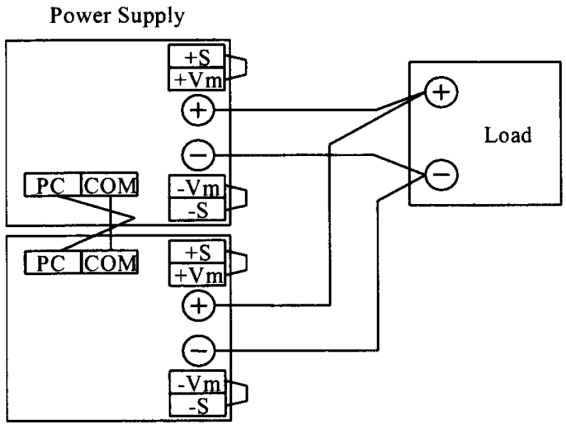

Figure 2.10: Parallel operation wiring from [1]

This effect can produce large voltage spikes as observed at the output of the supplies. These transient voltage spikes are able to trigger the power supplies' over-voltage protection, which in turn disables the supplies. This behavior is not desirable, as dynamic maneuvers often involve sudden changes in trajectory which trigger the regenerative mode of operation on the amplifiers. As shown in Fig. 2.11, a Schottky blocking diode was added at the output of the power supplies to prevent the over-voltage protection from being triggered. When the regenerative operation produces a voltage spike, the diode becomes reverse biased, which prevents the observation of the excess voltage by the supplies. This excess voltage charges the capacitor, which then begins to discharge as the voltage drops back to its normal level. If the voltage on this capacitor exceeds 88V, the amplifiers will be automatically shut down. An increased capacitance will be necessary to prevent this undesired behavior. To prevent diode breakdown under this most extreme voltage, the Schottky diode selected has a maximum reverse voltage of 50V.

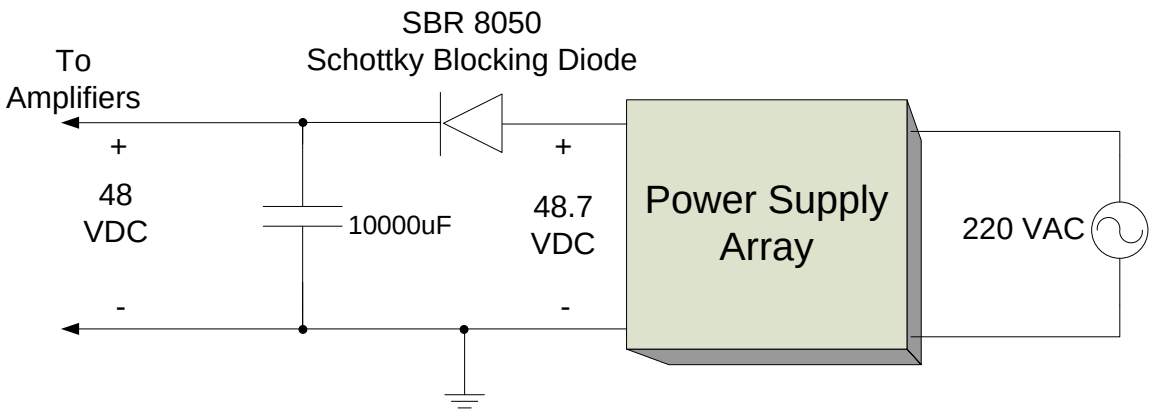

Figure 2.11: Blocking diode circuitry

2.8 Galil Motion Controller

The Galil DMC-4040 Motion Controller is the central component of the foundational control system and provides the interface to all high-level control. It has a RISC based 68000 processor at its core. The controller has 8 encoder inputs, 4 for use in feedback, and 4 auxiliary inputs. As mentioned previously, it has 8 optically-isolated digital inputs and 8 analog inputs which interface with a 12-bit ADC [2].

Work in [10] found an embedded K-Team system to be a light-weight solution for real-time control of a single leg. This setup employed four PIC microcontrollers for real-time control of the leg's two actuated degrees of freedom, and a 400MHz credit card sized Linux system to perform high-level motion planning. Yet, this system was significantly limited by its communication bus between the Linux and PIC components. Substantial work in [11] improved this communication through redesign of the protocol. Still, data feedback took 3.3ms with this platform. The Galil

motion controller, on the other hand, allows communication over a 100Mbs Ethernet connection, making communication delays negligible. Additionally, the Galil offers a simple-to-use two-letter command set with command execution times of $\approx 40\mu s$. The Galil also offers a variety of motion modes that were unavailable on the K-Team platform.

2.8.1 Motion Modes Available

Although the Galil offers a wide variety of motion modes, two modes were found most applicable for coordinated control of the articulated legs. The first of these modes, Independent Axis Positioning, provides independent motion for each axis based on trapezoidal velocity profiles. As a result, this mode of motion creates linear-parabolic blend motion profiles. The trajectory for each axis is characterized by its acceleration, maximum speed, deceleration, and final position. Thus, to produce coordinated motion of multiple joints, these parameters have to be calculated based on the desired time of motion, acceleration period, and deceleration period. During execution of a linear-parabolic blend position profile, the motion controller determines a new desired position every servo loop for input to a digital position feedback loop. This mode is most applicable when a trajectory can be specified which will not need to be dynamically modified during its execution.

The next of these modes, Contour Mode, allows an arbitrary position profile to be prescribed for all four axes. The contour is characterized by motor position increments each over a specific time interval. Within each specified time interval, the desired trajectory is assumed to have constant speed, which causes this mode to have abrupt changes in velocity at the endpoints of each contour segment. Still,

the flexibility of this mode allows the user to specify a portion of the contour, begin the motion, and then dynamically add segments to the contour. Thus, this mode is applicable when a complex trajectory is required, or when the trajectory needs to be dynamically retargeted during its execution.

2.8.2 Selection of PD Gains

Each of the motion modes discussed makes use of the Galil's digital PD position feedback loops. These feedback loops execute automatically on the motion controller at user-specified rates. The work discussed in this document has used .5 ms and 1 ms servo loop periods. In each of the modes discussed, a trajectory is inherently specified as a function of time. Each time the servo loop is executed on the Galil, the desired position for each axis is determined from this trajectory and compared to encoder feedback. The error in desired position is passed through a digital PD filter and ultimately converted to a PWM current command, which is passed to the amplifier.

Selection of the proper PD gains is imperative for performance of this position feedback system. Gains were initially tuned for the system on an unloaded motor and drive system. Due to the availability of graphical tuning software, provided by Galil, these gains were tuned iteratively by hand. The proportional gain was increased until overshoot to a step was observed, then the derivative gains was increased until the system became overdamped. This was repeated until further increases in the proportional gain or derivative created undesirable, small-amplitude, high-frequency oscillations in the motor at steady state. This produced a proportional gain on the motion controller of 186, and a derivative gain of 886. Physically, with respect to

an error in motor position prior to the 126:1 gear ratio, these correspond to gains of $K_p = 21.7$ A/rad and $K_d = .87$ A·s/rad. Figure 2.12 shows the step response of the resulting system to a step input of 100 counts.

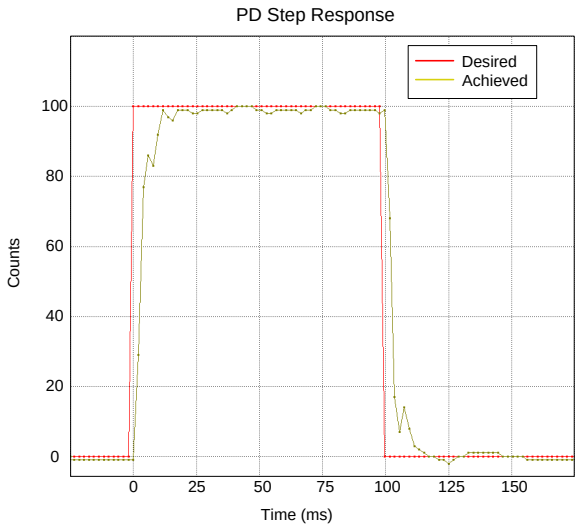

Figure 2.12: Motor position PD step response

2.8.3 Interface to Host

One of the main benefits of the Galil motion controller is its high-speed Ethernet communication with host machines. Galil provides a high-level communication library, for C++, to take advantage of this connection. Commands can be issued with this library which follow the two-letter command syntax used by on-board motion programs. This, for example, allows complex trajectories to be specified on the host, while contour mode commands issued over the Ethernet relay these trajectories to the motion controller.

Additionally the Galil can be set to periodically update the host with the its status. This status update, called a data record, contains information on the state of each axis' motion, the voltage of each analog input, and the state of each digital input. These data records will be used extensively in the following section.

2.9 Summary

This chapter has outlined the foundational components, electrical and mechanical, which provide the lowest level of control for the biped. A model was introduced to describe the kinematic configuration of the biped and its series-elastic actuators. An in depth look at the all sensors used in high-level control, from the boom encoders to link position sensing, has been presented. Finally, the central component of the foundation control system, the Galil motion controller, and its operation have been summarized and will be built upon in the following chapter.

CHAPTER 3

Design and Implementation of a Distributed Control System

3.1 Introduction

This chapter will discuss the software tools used to establish distributed real-time control of the bipedal system. As mentioned in the previous section, the high-speed nature of dynamic maneuvers requires control of the biped to happen on a millisecond time scale. While the foundational control system provides a relatively light-weight low-level control solution for the biped, its limited processing capabilities are often unable to perform computationally complex tasks on a strict time schedule. In particular, calculations of forward and inverse kinematics, which involve computations of trigonometric and inverse trigonometric functions, are unable to execute at necessary rates when implemented on the Galil motion controller. As a result, a distributed control structure is desired to meet real-time deadlines.

The addition of a high-speed host computer offers other advantages as well. First, high-level programming languages available on the host machine allow for a layered control strategy to be implemented. This allows higher-level control strategies to operate at increasing levels of competence, and thus to accomplish increasingly complex tasks [28]. Evidence suggests that biological systems employ a similar strategy, sequencing simple low-level motor primitives to create complex behaviors [29, 30]. To

this end, the software developed uses low-level motor primitives through state-based control strategies. The modular nature of the approach allows for further layers to be added in the future.

In addition, the host-computer provides a platform to interface with the experimenter through graphical feedback. A graphical user interface was developed as part of the distributed framework to provide quick configuration of the control environment and near real-time system state feedback. This feedback helps to debug the software and alerts the experimenter of system problems prior to their creation of a safety hazard in the hardware.

3.2 Overview of Main Design Concepts

The host software has been developed on a real-time Linux machine with the Qt development framework. The Qt framework is a cross-platform, application and user interface framework which provides programmers with a set of intuitive C++ APIs. For the distributed control software, Qt has mainly been used for its GUI and multi-threading classes. In addition to the Qt APIs, a GalilTools communication API was used to implement communication with the foundational control system. One of the major components of this communication is the periodic transmission of data records from the Galil to the host. With these data records, a snapshot of the Galil's current state is transmitted to the host every two servo loops. Then, based on this information, the host software can make high-level control decisions, such as the execution of a new motor primitive. These actions are distributed amongst two threads, while a third is used to update the GUI. Operating system calls are used to

designate the multi-threaded process as real-time. This gives all of the threads the highest scheduling priority possible on the dual-core host computer.

The high-level data flow diagram in Fig. 3.1 shows the role of each thread and the interaction between the threads. On this diagram, data records objects are shown in blue, thread instances are shown in gray, and periodic events are signified by a yellow clock. The two main global objects are the data record array and Galil connection object. The data record array is stored in memory so that a detailed log of the system state can be written to disk after time-critical control operations are complete. The Galil connection object provides a wrapper for the more complex GalilTools API.

Each of the three threads plays a specific role within the distributed control framework. The data thread's main role is to repackage the data record information sent by the Galil into a convenient data record object. This object type allows the programmer to request information about the system state, such as a motor angle or link velocity, without the need to perform conversions or numerical differentiation. All conversions, calculations, and storage are internal to the data record object, which encapsulates this complexity from the state monitor and GUI threads. Once the data record has been processed, the data thread adds the new object to the data record array and signals the state monitor thread that new data is available. The state monitor thread then checks the data record for any safety concerns and notifies the user through a log message display on the GUI. Any error messages from the Galil are also handled by the state monitor thread. If a high-level state machine is active, such as the state machine for a run or a jump, the data record is then passed to the state machine for high-level control decisions. The GUI thread updates the user interface at a specified frequency. Like the state monitor thread, it uses the most current data

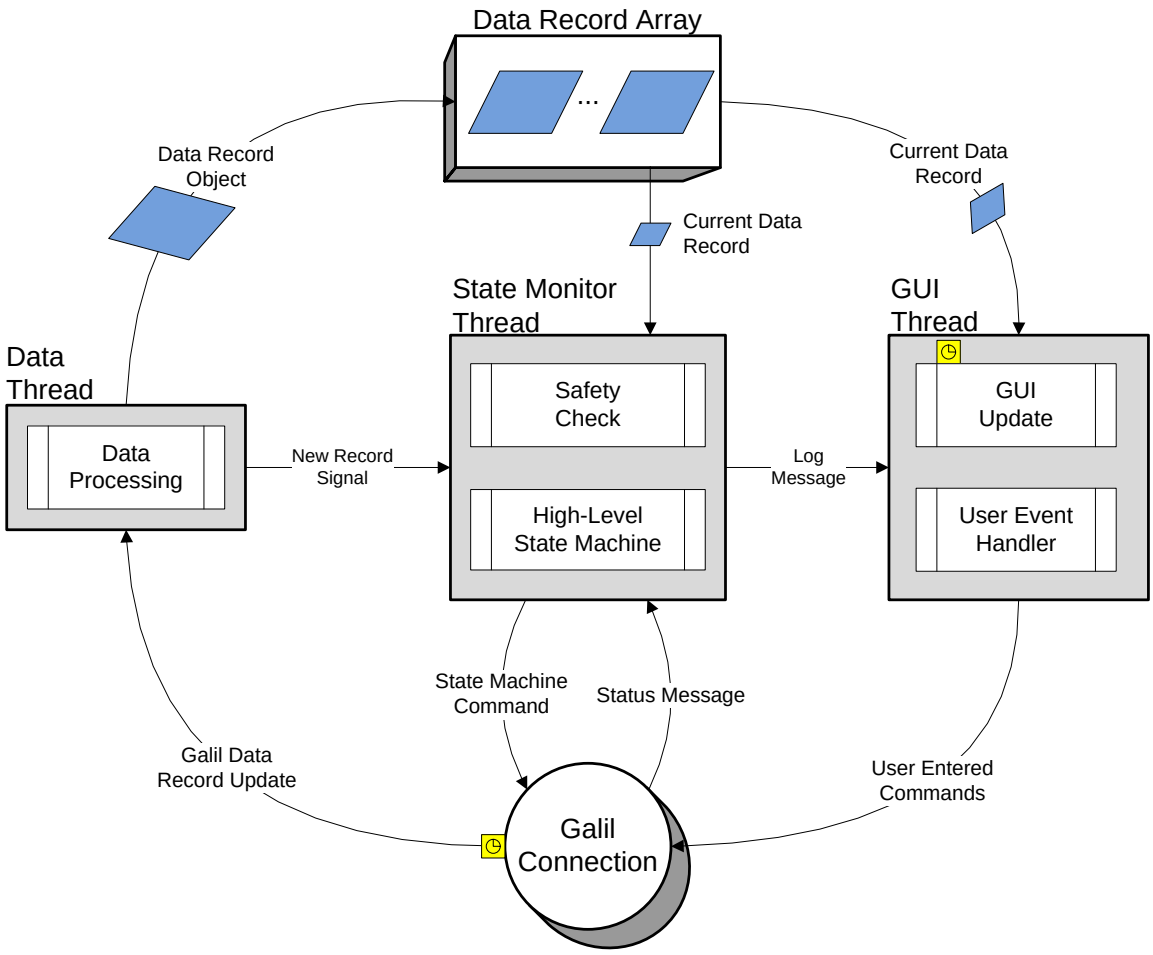

Figure 3.1: Host software data flow diagram

record for the user interface update. Additionally, this thread handles any user input from the GUI and directly issues commands to the Galil based on that input. The following sections will describe this interaction in more detail.

3.3 Data Processing Thread

The main role of the data thread is to package the Galil data records into data record objects. This thread populates record objects from Galil data record updates, numerically differentiates certain data, and applies filters when necessary.

3.3.1 Population of Data Records

Data record updates are sent by the Galil motion controller every two servo loops. The physical state information in these Galil data records uses a variety of units. For example, it uses encoder counts for encoder inputs, volts for analog inputs, or boolean values for digital inputs. Each of these require different conversion factors to give them physical meaning. Additionally, some information, such as the height of the left foot from the ground, requires a complex calculation from a combination of other data. The data record object provides a standard interface to all physical system data, referred to as readings, regardless of origin, necessary conversions, or complex calculations required. Internally, the data record class employs a static structure, called the reading type map, to fully describe each reading. The different reading types include DIRECT, for readings taken directly from the Galil updates, CONVERTED, for readings converted based on a linear equation from a DIRECT reading, and calculated, for readings that require complex calculation. This map also includes information specific to each type, such as a function pointer for readings that require complex calculation, or a conversion factor for converted readings. The function pointers used correspond to functions implemented within the data record class to perform computation.

In addition, some information, such as link velocities, requires numerical differentiation and filtering, which relies on data in multiple data records. This data, or any data that cannot be directly calculated from a single Galil data record update, must be entered specially. These types of readings are stored within the reading type map with a type of SPECIAL. As a result of this setup, the data thread only needs to supply the data record object with the Galil data record update and any SPECIAL data. Once the record is populated, the programmer may retrieve readings without regards to their origin. The following demonstrates how to query a data record object for a reading.

```
i=dataRec->getReading(LKNEE_MOT_CURRENT);
h=dataRec->getReading(LFOOT_HEIGHT);
```

All of the reading names, such as LKNEE_MOT_CURRENT or LFOOT_HEIGHT, are defined in an enumeration in the source file ReadingDefines.h. The enumeration is used to provide constant access time to all the array data structures within the data record object, instead of O(log(n)) access time to a string indexed map. In addition, the Galil API's string indexed data record structure is converted to constant access time array for use in future calculations.

3.3.2 Digital Filtering

Many physical states, such as motor or link velocities, are not directly sensed in hardware, yet are required by high-level control strategies. Examples of use are the detection of bottom and top of flight during execution of a jump. These readings require numerical differentiation, followed by a low-pass filter to remove the high-frequency noise amplification inherent in numerical differentiation. Numerical differentiation is accomplished through a backward difference method as shown below, where T is the

sampling frequency. In the case of the data thread, the sampling period is double the servo loop time.

$$\frac{dy}{dt}[n] \approx \frac{y[n] - y[n-1]}{T} \tag{3.1}$$

Digital filters were designed by conversion of the design problem to an analog design problem. This approach is borrowed from [31]. After standard analog pole-zero formulas were used to generate a transfer function, the analog filter was then transferred back to a digital filter. The conversion of a digital filter design problem to an analog one is accomplished through the use of the bilinear transform, shown in Fig. 3.2. This transformation establishes a invertible map between the $+j\omega$ axis in the s-plane to the upper half of the unit circle in the z-plane[2]. The transformation is shown below in Eq. 3.2 and creates a relationship between analog and digital frequencies as shown in Eq. 3.3. Its application to convert generic transfer function from an analog to a digital one is shown in Eq. 3.4.

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{3.2}$$

$$\omega_a = \frac{2}{T}\tan(\omega_d \frac{T}{2}) = \frac{2}{T}\tan(\pi f_d T) \tag{3.3}$$

$$H_a(s) \leftrightarrow H_a\left(\frac{2}{T}\frac{z-1}{z+1}\right) = H_d(z) \tag{3.4}$$

That is, if $H_a(s)$ is converted to $H_d(z)$ through the use of the bilinear transform, the digital frequency response from 0 Hz to the Nyquist rate ($f_d = 1/(2T)$) of $H_d(z)$, will attain all values obtained by $H_a(s)$ from 0 to $+\infty$ Hz. Put differently, the entire analog frequency response is compressed to fit within digital frequencies from

[2]More generally the transformation establishes a invertible map between the entire LHP in the s-plane, to the area enclosed by the unit circle in the z-plane.

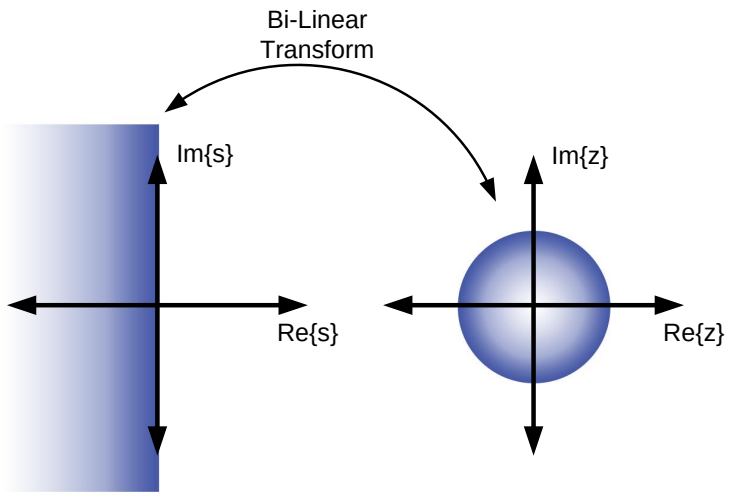

Figure 3.2: Bilinear transform

0 Hz to half the sampling frequency. The non-linear nature of this compression is commonly referred to as frequency warping. To account for this warping, the desired digital cutoff frequency is used with Eq. 3.3 to find the corresponding analog cutoff frequency for design. Once standard analog filter design produces an analog transfer function, Eq. 3.4 is applied to produce the final digital transfer function. This process was implemented for Butterworth, Chebychev, and Elliptic filters. Details on their design specifics can be found in Appendix B.

A graphical interface was created to tune filter parameters as shown in Fig. 3.3. The top two graphs show the time domain performance of the filter through step and ramp response graphs. The steady state ramp response is used to estimate the delay for the filter[3]. The bottom graphs show the frequency domain performance of the

[3]Since the IIR filters designed do not have a linear phase response, this estimate should be used as such. Delays are frequency dependent and should be investigated more closely when precise delays need to be known.

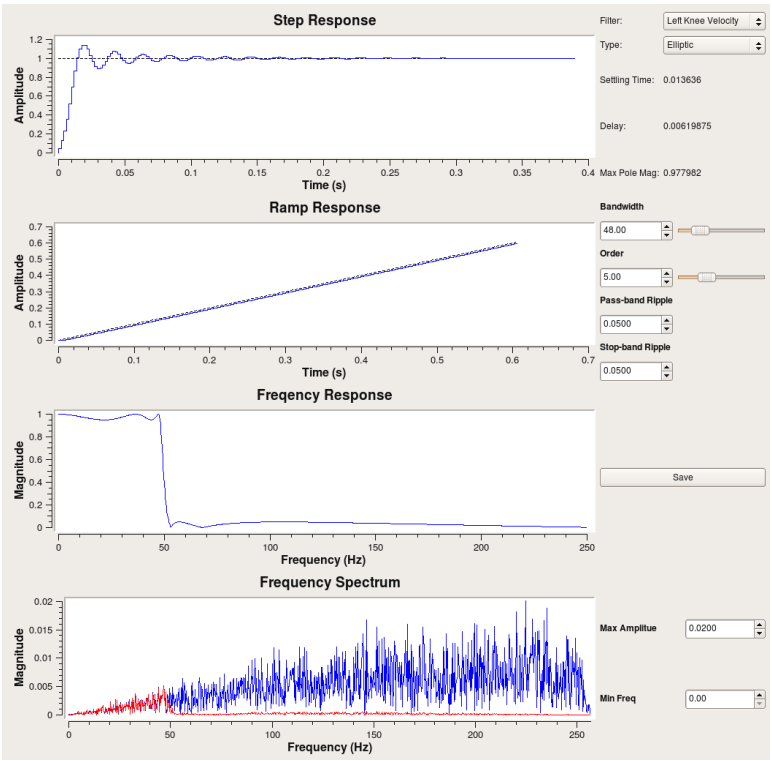

Figure 3.3: Filter configuration user interface

filter. The first shows the magnitude response of the digital filter with linear axis scales. The bottom shows a 2048-point FFT of the real-time filtered and unfiltered data. Use of this GUI provides an less error-prone alternative to the manual alteration of text based configuration files as suggested in [4].

3.3.3 Inter-Thread Communication

Once the most recent data record object is processed, a reference to the data record is published to data record array, and the state monitor thread is notified of the new record. Since this data can be accessed by multiple threads, special care was taken to ensure data integrity. First, a simple mutex was employed to protect access

to the data record array. This mutex prevents two threads from modifying the array at the same time. Second, a read-write mutex was employed to protect threads that read from the most current data record. A read-write mutex was used since multiple readers (the state monitor and GUI threads) may need to access data at the same time, a feature that is not allowed with standard mutexes. This approach allows the threads to share access to data, as opposed to time consuming copies being made for each thread.

The other method of inter-thread communication employed by the data thread is an inter-thread signal, implemented as a wait condition. Each time the state monitor thread completes its examination of a data record, it waits for a new one. The use of a wait condition causes the state monitor thread to be woken up by a software interrupt. The data thread triggers this software interrupt through a wakeAll method on the wait condition object. The code below illustrates the state monitor and data thread calls as part of this process.

```
\\Wait for new data (State Monitor Thread)
newDataCondition.wait();

\\Signal that new data is available (Data Thread)
newDataCondition.wakeAll();
```

This method also ensures that the state monitor thread, which consumes the data records, does not fall behind the data thread, which produces the data records. As opposed to other producer-consumer constructs, this method will cause the state monitor thread to skip an unseen data record if a new one is available. This ensures that control decisions are always made on the most current state information available.

3.4 State Monitoring Thread

The role of the state monitor thread is to monitor the safety of the biped, and to execute high-level control strategies as needed. The state monitor thread retrieves the most current data record object upon reception of a software interrupt which indicates the availability of new data. It also checks for any log messages sent by the Galil to indicate errors or state change information. This data is used in combination by the safety routines and the high-level state based controllers.

3.4.1 Safety Measures Implemented

Current safety measures implemented include joint limit detection and actuator torque limit detection. Joint limit detection and correction is handled by the Galil during periods of closed-loop control. Commands for this operation are stored on the motion controller in the #SAFTEY routine. Software joint limits are defined as within 5° of the physical hardstops. When the software joint limits are exceeded, the low-level safety routine quickly moves the biped to its home configuration, where $\theta_h = -30°$ and $\theta_s = 30°$. The Galil sends an error message to the host to notify the operator of this event. This notification is ultimately handled by the state monitor thread.

In periods of open-loop control, joint limit detection is sensed on the host and the operator is notified. No corrective actions are taken, as the high-level control strategies are ultimately responsible to keep the biped away from these regions of operation. The same holds true for actuator deflection limits. The SEA springs have limits on their deflection before permanent physical damage results. The actuator

deflection warnings are provided at deflections of 1.9 rad, well below the designed limit of 2.167 rad.

3.4.2 Use of State Based Controllers

Previous work, [10, 32], has shown success with the decomposition of dynamic maneuvers into multiple low-level motor primitives. Types of motor primitives may include, open-loop current control, a closed-loop position trajectory, or the freeing of a joint from any control. This technique has biological motivation as well, as sources [29, 30] have shown that biological systems combine simple motion primitives to produce more complicated motion.

Since each dynamic maneuver will employ its own state machine, particular care was taken to ensure that the implementation of each state machine was decoupled from a standard state machine interface. A virtual state machine class was implemented to provide all functionality common to each state machine. This class also provides the specification of a virtual function, checkState, to be implemented by any inheriting classes. The implementation of this function provides a state machine's control policy based on a data record input. Any messages sent by the Galil are also passed to this function to allow the state machine to handle errors. As a result of this design, the switch from a state machine for a jump to a state machine for a walk requires a one line change of code as shown below. The call by the state monitor thread to perform high-level state-based control does not change. This represents one of the major contributions of this work, as new dynamic maneuvers will require only the implementation of a state machine class to interface with the distributed control framework.

```
\\Create a Jumping or Walking State Machine
StateMachine * mainStateMachine = new JumpingStateMachine();
\\Or
StateMachine * mainStateMachine = new WalkingStateMachine();

\\Pass a data record to the state machine for control actions
mainStateMachine->checkState(galilMessage,currentDataRecord);
```

3.5 Graphical Feedback

The final thread in the distributed control software is a thread for user interaction with the GUI. Both periodic GUI updates and user triggered events are handled by this thread. The GUI for the jumping state machine can be found in Fig. 3.4. The first part of the GUI, signified by a 1 on the figure, contains a variety of numeric and boolean indicators for system state information. Information such as the biped's current height, each actuator's SEA deflection, the status of each foot sensor, and the servo status of each motor are all shown on this pane. The GUI thread uses a large data structure to store each indicator and its associated data record reading. Modification of the GUI layout can be accomplished through addition to or rearrangement of this data structure.

The second part of the GUI shows the status of the state-based control strategy. The state machine display contains all possible states and highlights the current state with a gray background. The state machine can be activated or deactivated through a control on the right side of this display. The implementation of this display is in the virtual state machine class and thus does not have to be repeated for future maneuvers. The third section provides a 3-D view of the system state through the use of a OpenGL extension to the Qt framework. Dark rectangles near each link indicate

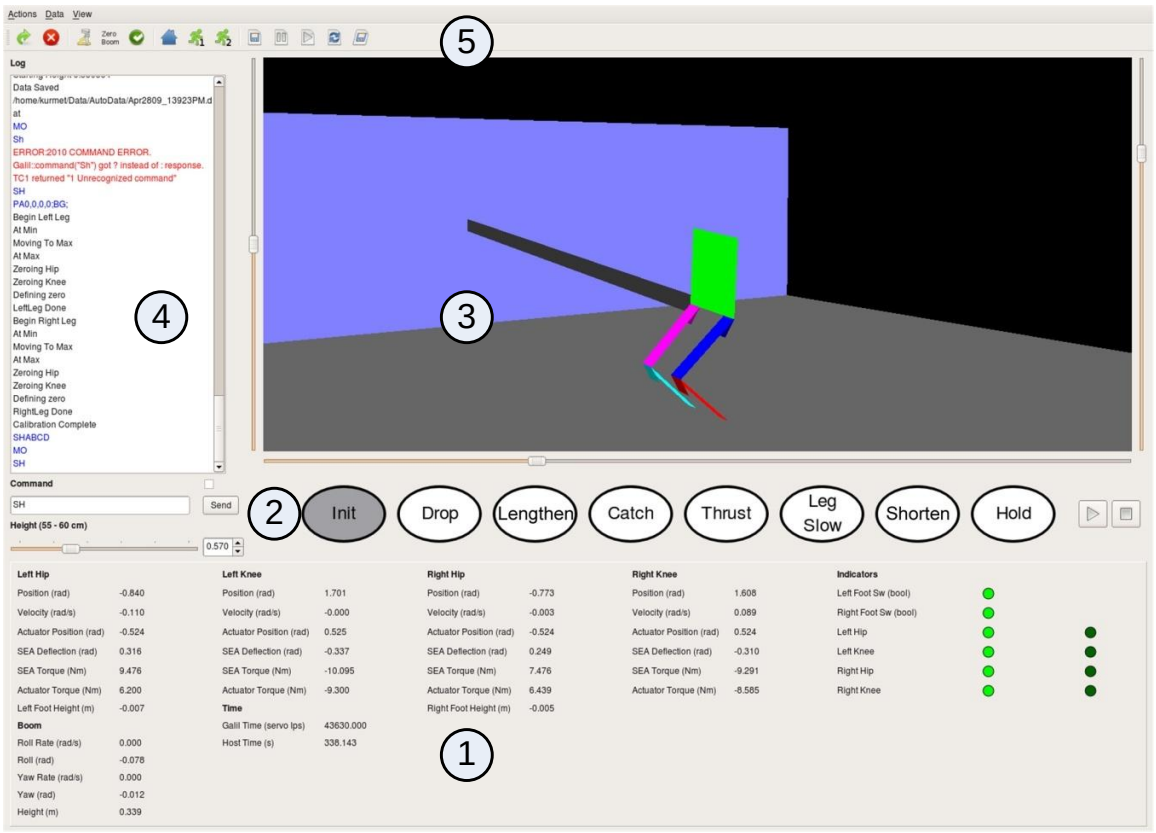

Figure 3.4: Graphical user interface for host software. 1) sensor readings 2) state machine control state 3) sensed kinematic configuration 4) debug and foundational control terminal 5) quick access toolbar

the position of the motor and can be used to visualize the SEA deflections. The next section, on the left, gives the user a terminal to enter commands directly to the Galil. Any response from the Galil is shown in the scrollable log display in this pane. The log box is also used to display any user debug messages, or any unsolicited messages from the Galil. This section also contains any user input widgets for the high-level control, such as the desired height slider for a jump. The final portion of the GUI, the toolbar, provides a quick interface to many common tasks. These tasks include

the ability to zero the boom, save a detailed log of the system state, calibrate the system, or to perform a leg cycle.

All information on the GUI is updated at a user specified rate. During execution of a jump, for example, the GUI is updated twice per second. Although the update process is CPU intensive, this low update frequency does not hinder the performance of the other, more important, threads. Still, if the host software is ever moved to a single core machine, the possibility of missed control loops due to the GUI should be investigated.

Benefits

Other than ease of use brought about by the GUI, the graphical feedback provides distinct advantages for safety and the ability to debug the system. The motor status indicators on the first pane indicate if the Galil is in closed-loop position control for each axis. If the emergency stop switch has been engaged, care should be taken to monitor these inputs before power is returned to the system. Failure to do so could result in a powerful jerk of the motors to the desired position. Additionally, the foot switch indicators should be watched, as a jammed foot switch can result in poor timed state transitions which rely on ground contact. The 3-D view of the system can also be used to quickly isolate any sensor issues. In terms of debugging, the log display can be used throughout the code with the call below. It is commonly used to provide immediate feedback for information such as jump height, or the time spent in communication for a specific motor primitive. Any message sent from the Galil will appear in this display as well, which facilitates low-level development on the Galil.

```
guiUpdateThread->sendLog("Hello World");
```

3.6 Summary

This section has presented the details of the distributed control software present on the Linux host machine. This section has highlighted the features and implementation of this software and outlined all major design decisions. The modular nature of the framework has been stressed, as the ease of integration for new dynamic maneuvers highlights on of the main contributions of this work. The role of the data thread, state monitor thread, and GUI thread have been discussed, as well as their inter-thread communication strategies which ensure that real-time control is maintained.

CHAPTER 4

Jump Controller

4.1 Introduction

In the study of dynamic maneuvers, the standing jump is one of the most basic. Although the jump does experience extended periods of ground contact, 200-250 ms in a typical jump for KURMET, the high-power thrusts and complex energy exchange during contact are fundamental to the execution of any other type of maneuver. Thus, the jump provides a good starting point for the study of dynamic maneuvers. Previous work has produced a single-leg capable of a high-performance jump [10]. A state machine structure, similar to those discussed previously, was employed on an embedded Linux machine. This setup ultimately was able to produce 15 consecutive jumps in hardware [11].

KURMET's motion on the boom is much less constrained than in the previous Hopper setup which increases the complexity of the control problem. To address this additional complexity, work by Hester in [20] uses an intelligent fuzzy controller for supervisory control of the biped's jump height and forward velocity. A state-based controller for a jump is detailed in his work. The state machine is cyclic and uses the fuzzy controller at the top of each jump to select appropriate parameters for the next jump. The control structure was developed in *RobotBuilder*, a graphical package for

the dynamic simulation of robotic systems [33]. Repeated simulation jumps were used to train the rule base for the fuzzy controller, and to demonstrate its use in repeated jumps. The implementation of the state-based control strategy for hardware and a brief description of results will be presented in the following sections.

4.2 Description of States

Figure 4.1 shows the state transition diagram for the jump controller. The supervisory fuzzy controller is executed at top of flight and returns the touchdown configuration, catch currents, and thrust currents for the next jump. This state structure, similar to that presented in [20], differs in the addition of the INIT and DROP states. To start a jump in simulation, the system is placed at an initial height with no velocity. In physical experiments, the experimenter must hold the biped in the air and drop it to begin a test. In the INIT state, commands are issued to the Galil to position the feet in the TOF configuration. In the DROP state, the system waits for the feet to be in position, and then provides an audio cue to indicate that the system is ready to be dropped. The transition into the cyclic portion of the state machine then occurs when the boom pitch velocity has reached a certain threshold. This threshold was set experimentally to -0.05 rad/sec, which prevents false transitions from small boom fluctuations while the system is held.

The rest of the state machine follows the setup outlined in [20] and will be briefly outlined. The LENGTHEN state moves the feet into their touchdown configuration. For the transition into LENGTHEN, motor trajectory parameters are calculated on the host machine and sent to the Galil. The time to touchdown is estimated based on simple ballistic physics. The motor trajectory is executed in half of this estimated

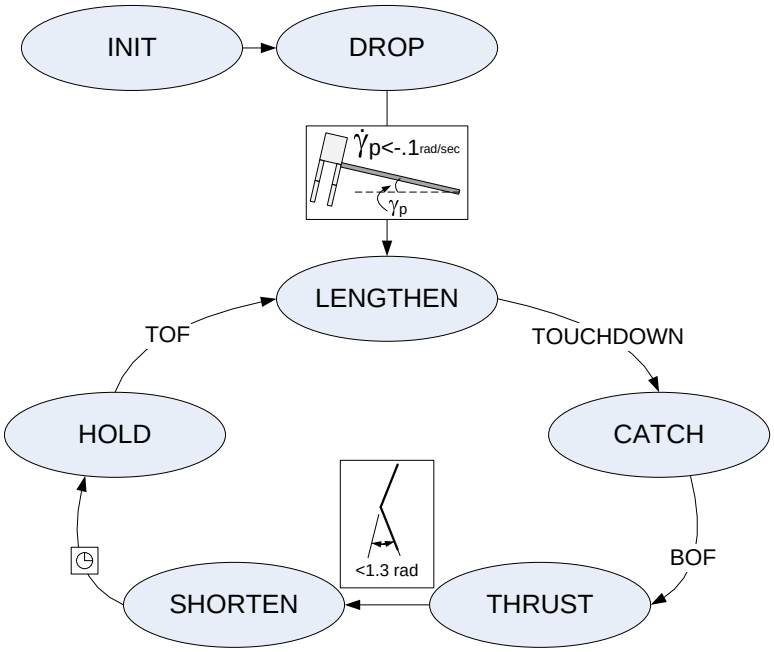

Figure 4.1: State transition diagram for the jump controller

time to ensure that the foot is in position upon ground contact. The trajectory uses a triangular velocity profile to minimize system accelerations and decelerations. This is important, as high accelerations and decelerations can cause SEA deflections during flight which lead to imprecise foot placement upon touchdown. Let t be the trajectory time, h the initial height, a the acceleration, d the deceleration, v the max speed, and $\Delta\theta$ the change in motor position. The trajectory parameters are calculated as follows:

$$t = .5\sqrt{2(h - .3 \text{ m}) \ / \ 9.81 \ \frac{\text{m}}{\text{s}^2}} \tag{4.1}$$

$$a = d = 4\frac{\Delta\theta}{t^2} \tag{4.2}$$

$$v = \frac{at}{2} \tag{4.3}$$

The transition to CATCH occurs upon touchdown, and was significantly changed for hardware implementation. The transition occurs when the foot height is calculated to be less than 5 mm. This provides a more robust solution than use of the foot switch, as inertial loads or stuck foot switches can cause faulty contact readings. Upon touchdown, open-loop catch currents are commanded to deliver energy to the SEAs and to keep the links away from the hardstops. Commands are issued by the host to disable the PD position control and initiate open-loop current control. More specifically, these commands set the Galil proportional and derivative gains to 0, and command an offset current.

The transition to THRUST was modified for hardware as well. This transition occurs at the bottom of flight (BOF), when the angle between the thigh and shank begins to decrease. Mathematically, this corresponds to the case when $\dot{\theta}_s - \dot{\theta}_t < 0$. Due to noise introduced from numerical differentiation, this transition can be falsely triggered. As a result, similar to in the DROP primitive, a conservative threshold is introduced to prevent false transitions. The threshold was experimentally set to -0.1 rad/sec. It prevents false transitions and adds minimal delay for the trigger of correct transitions. Additionally, a minimum time of CATCH was implemented to further prevent false transitions. As a result, 70 ms elapses in CATCH before the transition criteria are examined. Once bottom of flight is sensed, command currents are modified to the THRUST currents as specified by the fuzzy controller.

While termination criteria for THRUST was not modified for implementation, particular care was taken to ensure a fast transition to SHORTEN. For the transition to SHORTEN, work in [20] found that thrust currents need to be cut-off prior to liftoff in order to avoid contact with link hardstops. More specifically, to avoid

these hardstops for all jumps within the fuzzy's range, thrust had to cease when $\theta_s - \theta_t < 1.3$ rad. This result is used in hardware without modification. Unlike the LENGTHEN primitive, where a precise start to the motor trajectories is not critical, immediate PD control of the motors is necessary for the SHORTEN primitive to prevent contact with the hard stops. To optimize transition latency, a minimal number of parameters are passed to the Galil at the desired transition time. The final motor positions for the SHORTEN primitive are statically known, and are sent to the Galil during INIT. The current height and a closed-loop flag are then sent to the Galil at the time of transition. All other trajectory parameters are calculated on-board, and the trajectory is begun within 2 ms. The trajectory duration is based off a fraction of the time to reach the desired height. The trajectory is one third acceleration, two thirds deceleration to provide additional deceleration time. This extra time is needed since deceleration during shortening has a tendency to deflect the SEAs, which can cause imprecise foot positions upon landing. Let t be the trajectory time, h_d the desired height, a the acceleration, d the deceleration, v the max speed, and $\Delta\theta$ the change in motor position. The trajectory parameters are calculated as follows:

$$t = .6\sqrt{2(h_d - .3 \text{ m}) \ / \ 9.81 \ \frac{\text{m}}{\text{s}^2}} \tag{4.4}$$

$$v = 2\Delta\theta/t \tag{4.5}$$

$$d = 3\Delta\theta/t^2 \tag{4.6}$$

$$a = 6\Delta\theta/t^2 \tag{4.7}$$

Once top of flight is sensed, the host calculates an average yaw velocity since the end of thrust for use as an input to the fuzzy controller. Once the fuzzy controller has calculated parameters for the next jump, the state machine proceeds to the LENGTHEN

state once more. After a desired number of jumps have been completed, the state machine exits from the LENGTHEN primitive and automatically logs the system data record array to disk.

4.3 Results

Figure 4.2 shows the hip height over time for a 60 cm jump followed by a 57 cm jump. Figure 4.3 shows the actuator and link angles for the left leg over the same series of two jumps. The responsiveness of the control can be observed qualitatively at impact, marked by the deviation of the link angle from the motor angle. The injection of catch currents appears almost instantaneous, as both the hip and knee motors begin to deflect 2 ms after the observed touchdown.

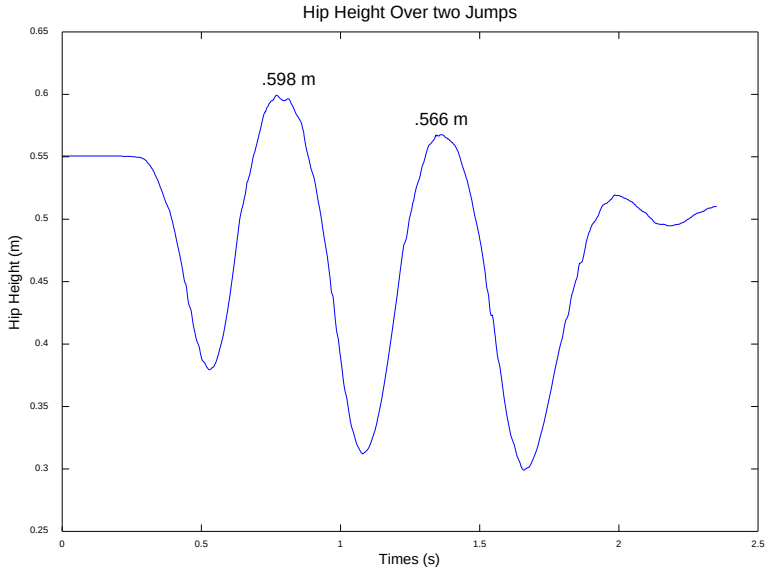

Figure 4.2: Hip height over two jumps

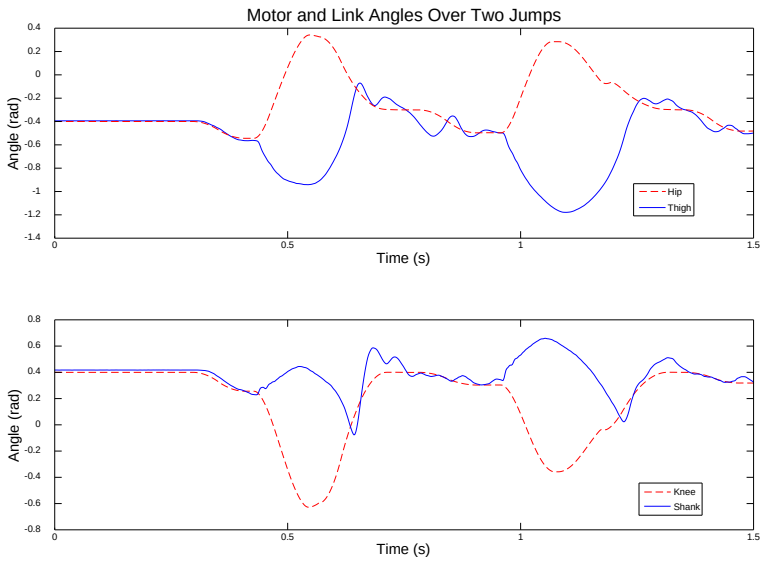

Figure 4.3: Motor and link angles over two jumps

Table 4.1 presents the typical delays for each state transition. As shown, the state transitions happen on a precise time schedule when necessary. These time delays include communication, sensing, and calculation delays. Thus, they represent the delay between when the motor primitive would ideally start and when it actually starts. The transition from DROP to LENGTHEN is delayed due to the boom pitch velocity filter delay and the conservative threshold on velocity. The transition to SHORTEN is delayed primarily by the limited communication which takes place. Finally, the transition from HOLD to LENGTHEN is delayed mainly due to boom pitch velocity filter delay, which accounts for 6 ms of the total delay.

Transition From	Transition To	Ideal Transition Criterion	Delay from Ideal
DROP	LENGTHEN	$\dot{\gamma_p} < 0$ rad/s	$\approx$ 50ms
LENGTHEN	CATCH	Touchdown	<2ms
CATCH	THRUST	$\dot{\theta_s} - \dot{\theta_t} < 0$ rad/s	<2ms
THRUST	SHORTEN	$\theta_s - \theta_t < 1.3$ rad	4ms
HOLD	LENGTHEN	$\dot{\gamma_p} < 0$ rad/s	8ms

Table 4.1: Typical time delays for jumping state transitions

4.4 Summary

This chapter has presented the application of the control framework for the control of repetitive jumping. The jumping state machine has been presented, with particular details on the implementation details of state transitions. The results have shown the success of the control system for a high-performance, 60 cm jump. Crucial state transitions show millisecond-scale time delays which demonstrate the responsiveness of the real-time control framework.

CHAPTER 5

Walking Controller

5.1 Introduction

This chapter will outline the design and implementation of a walking controller for the bipedal system. Although walking may not exhibit many of the high-power characteristics of other dynamic maneuvers, its development is a step towards the execution of a dynamic run in hardware. Previous work in the LAB^2 at Ohio State has studied dynamic bipedal walking extensively [18]. These studies took place on a robot name ERNIE, which was designed specifically for walking. In contrast to ERNIE, design emphasis for KURMET was placed on its ability to execute high-performance dynamic maneuvers. One of the design differences can be found in KURMET's series-elastic actuators, which provide joint torques through spring deflections. While these soft springs in KURMET's SEAs provide benefits for the execution of many dynamic maneuvers, they present a challenge for the control of less aggressive movements such as a walk. One of the major challenges is to control the SEA deflection under ground contact loads while still controlling the link position. This second part is complicated by the link positions' slow yet oscillatory natural response to changes in motor position while the leg is loaded. In order to address the complexity presented by the SEAs, the unactuated DOF at the hip was removed for this work. This prevents the biped

from falling over, and leaves the maintenance of dynamic stability for the bipedal platform to be addressed in further work.

A supervisory speed controller was developed to provide full gait parametrization based on the desired speed. This supervisory controller, like the supervisory fuzzy controller for jumping, relies on a low-level state-based controller for execution of the motion. As opposed to the jumping state machine, where state transitions are based on feedback from the system, the walking-state machine is locked into the kinematic cycle phase of the periodic gait. The cycle phase, along with the gait parameters, can be used to fully characterize the desired configuration of the biped. This will be discussed more fully in the next section. As mentioned previously, the presence of series elasticity in the actuators complicates the execution of motion during ground contact. A gravity compensation strategy was developed to support the weight of the robot through controlled deflection of its SEAs.

Finally, the control strategy was both simulated in *RobotBuilder* and implemented in hardware. Simulation has demonstrated the control approach's applicability over a wide range of speeds and shown its ability to transition from forward to backward motion. Initial experiments with the control in hardware have produced forward walking gaits which perform at fixed speeds from .15 m/s to .45 m/s.

5.2 Gait Parametrization and Kinematic Cycle Phase

Prior to discussion of the supervisory speed controller, notation will be developed to describe the desired gait of the biped. The walking gait is a periodic and consists of phases of single and double support. A leg that is on the ground will be referred to as a support leg, while one in flight will be referred as a swing leg. The time that

a swing leg is in flight will be referred to as the return time, τ. Table 5.1 describes the gait parameters as presented in [34]. Relationships between the gait parameters can be found in Eqs. 5.1, 5.2, and 5.3.

$$\eta = \beta\lambda \tag{5.1}$$

$$u = \lambda/T \tag{5.2}$$

$$T = \lambda/u = \eta/(\beta u) \tag{5.3}$$

Parameter	**Symbol**	**Description**
Stride Length	λ	the distance the body is translated in one locomotion cycle of the gait
Stroke	η	the distance the body is translated when a leg is in the support phase
Period	T	the time required for one complete locomotion cycle of the gait
Duty Factor	β	the fraction of a a locomotion cycle that each leg spends in the support phase
Velocity	u	the speed at which the body is translated

Table 5.1: Description of gait parameters for walking

From these relationships it can be observed that the gait is fully parameterized by the selection of: 2 of β, λ, or η, and 1 of u or T. For this work, the speed will be input by the user, while the duty factor and stride length will be selected by the supervisory controller. The kinematic cycle phase, ϕ, can now be defined as the distance by which the body has translated since the last placement of the right leg, normalized to the stride length. That is, as ϕ runs from 0 to 1, the gait will progress

through one complete locomotion cycle. Figure 5.1 shows the placement and liftoff events for the left and right feet with relation to the kinematic cycle phase and duty factor.

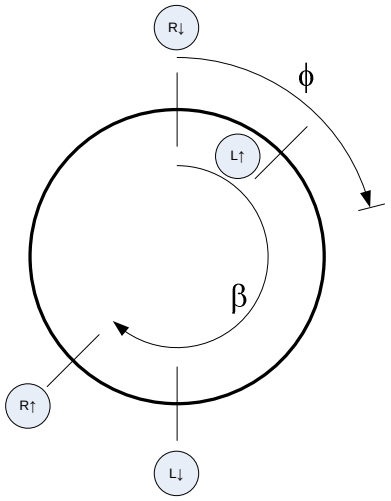

Figure 5.1: Foot placement and liftoff events as they relate to the kinematic cycle phase and duty factor

5.3 Supervisory Speed Controller

The supervisory speed controller was designed to select the stride length and duty factor of the gait based on a desired velocity. A number of restrictions were placed on the gait parameters to achieve reasonable behavior over a wide range of speeds. Three ranges of operation were identified for the supervisory controller: high-speed operation, mid-range operation, and high-duty factor operation.

High-speed operation is defined as operation with $|u| \geq u_{hs} = .25$ m/s. This threshold was tuned experimentally. At high speeds, the transition of support from

one leg to the other happens very quickly. When $\beta = .5$, this transition would theoretically happen instantaneously. In the physical system, weight support is provided by joint torques due to SEA deflection. An instantaneous change in this deflection is physically impossible, and has the tendency to incite unwanted oscillations in the link positions. As a result, a minimum duty factor greater than .5 was selected to allow time for a smooth transition of support. Additionally, the stride length is fixed at an upper limit, as longer strides produce leg angles further from singularity that require excessive joint torques. These limits were qualitatively tuned through experimentation and are summarized in Table 5.2.

High-duty factor operation is defined as operation at low speeds with $|u| \leq u_{hd}$. The speed threshold for high-duty factor operation comes from a upper restriction on the return time, denoted τ_{max}, of 1 second and a experimentally tuned duty factor threshold, β_{hd} of .8. The calculation for u_{hd} is shown in Eq. 5.4. At all speeds below u_{hd}, the duty factor of the gait is increased to maintain a return time of τ_{max} and a stride length of λ_{min}. This prevents the gait from using long periods of single support, during which the system would have to balance on one leg. Although balance is not an issue for the setup used, if dynamic walking is to be studied in the future, this restriction could prove to be more helpful.

$$u_{hd} = \frac{(1 - \beta_{hd})\lambda_{min}}{\tau_{max}} \tag{5.4}$$

Between these two modes exists a mode of mid-range speed operation. Over this range of speeds, from u_{hd} to u_{hs}, the stride length and duty factor vary linearly with speed. The duty factor varies from β_{hd} to β_{min} over this range, while the stride length varies from λ_{min} to λ_{max}. A summary of the gait parametrization for each mode of operation can be found in Table 5.3.

Parameter	Lower Limit	Upper Limit
Stride Length	$\lambda_{min} = .12$ m	$\lambda_{max} = .37$ m
Return Time	N/A	$\tau_{max} = 1$ s
Duty Factor	$\beta_{min} = .53$	N/A
Velocity	$u_{min} = -.5$ m/s	$u_{max} = .5$ m/s

Table 5.2: Description of gait parameter limits for walking

Mode	Velocity Range	Parameters
High Speed	$u_{hs} \leq \|u\| \leq u_{max}$	$\beta = \beta_{min}$ $\lambda = \lambda_{max}$
Mid-Range	$u_{hd} < \|u\| < u_{hs}$	$\beta = \beta_{hd} - (\beta_{hd} - \beta_{min})\frac{\|u\| - u_{hd}}{u_{hs} - u_{hd}}$ $\lambda = \lambda_{min} + (\lambda_{max} - \lambda_{min})\frac{\|u\| - u_{hd}}{u_{hs} - u_{hd}}$
High Duty Factor	$\|u\| \leq u_{hd}$	$\beta = 1 - \frac{\|u\|\tau_{max}}{\lambda_{min}}$ $\lambda = \lambda_{min}$

Table 5.3: Gait parameter calculations for walking

5.4 Motion Planning and Low-Level Control Strategy

Once the gait parameters have been selected, a lower-level of layered control uses the parameters to plan motor trajectories. This relationship is shown in the walking control diagram in Fig. 5.2. Link trajectories are calculated by a motion planner and inverse kinematics. These link trajectories, coupled with desired SEA deflections, determined by a gravity compensation algorithm, determine the desired motor trajectories to be tracked by the foundational control system.

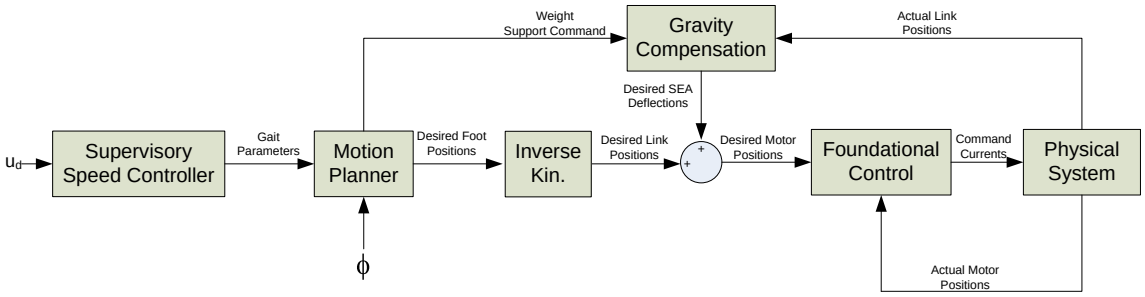

Figure 5.2: Overall block diagram for the walking control strategy

5.4.1 Motion Planning

At a closer look, the motion planner uses the kinematic cycle phase along with the gait parameters to specify the desired link positions. The cycle phase alone is used to command weight compensation for each leg. To accomplish this, the motion planner first decodes the cycle phase to determine the motion state as shown in Fig. 5.3. Once the state is known, foot trajectories are qualified based on the gait parameters. The following sub-sections will outline the specifics of these trajectories and the weight compensation commands.

Double Support Phases

In periods of double support, when both feet are on the ground, the system must transfer its weight support from its back leg to its front leg. Additionally, it must continue to move forward at a constant velocity. To provide a transfer of weight support, the motion planner uses a cubic spline for the commanded weight support for each leg. That is, for the front foot, the weight support command starts at 0 N, with a time derivative of 0 N/s. Over the course of double support, its weight

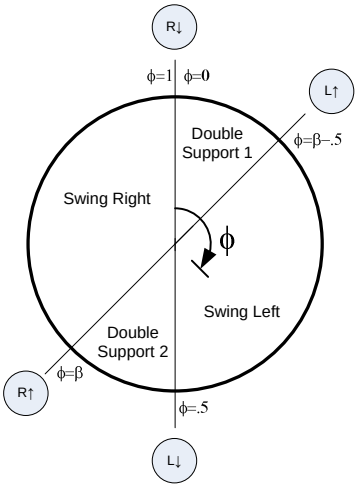

Figure 5.3: Gait states implied by the kinematic cycle phase

support command increases based on a cubic spline trajectory and ends at full weight support (118.6 N), with a time derivative of 0 N/s. This smooth change of support prevents step discontinuities in SEA deflection which incite unwanted oscillations in the system. Over the period of double support, the back leg's weight support command is symmetric to the front's with respect to time.

To provide a constant forward velocity of the body, v, with the feet on the ground, each foot must move with a velocity $-v$ with respect to the body. Thus, during support, the desired foot position moves with a constant velocity in the +x-direction while the y-position of the foot is held constant. Table 5.4 shows the initial and final x-positions of the foot during support. The y-position of each foot is calculated according to Eq. 5.5 in order to touchdown at configuration which is close to singular.

$$y_{foot} = -\sqrt{(.499 \text{ m})^2 + \left(\frac{\eta}{2}\right)^2} \tag{5.5}$$

Motion State	**Time in State**	**L. Foot** (x_o, x_f)	**R. Foot** (x_o, x_f)
Support 1	$(\beta - .5)T$	$(\frac{\lambda-\eta}{2}, \frac{\eta}{2})$	$(-\frac{\eta}{2}, -\frac{\lambda-\eta}{2})$
Swing Left	$(1-\beta)T$	-	$(-\frac{\lambda-\eta}{2}, \frac{\lambda-\eta}{2})$
Support 2	$(\beta - .5)T$	$(-\frac{\eta}{2}, -\frac{\lambda-\eta}{2})$	$(\frac{\lambda-\eta}{2}, \frac{\eta}{2})$
Swing Right	$(1-\beta)T$	$(-\frac{\lambda-\eta}{2}, \frac{\lambda-\eta}{2})$	-

Table 5.4: Support foot locations for each motion state

Single Support Phases

In periods of single support, the support leg's motion is simple, while the swing leg's motion is rather complex. The support leg is commanded full weight support and continues to move its foot with constant velocity in the +x-direction as shown in Table 5.4. The swing leg uses a series of five spline trajectories to ensure a continuously differentiable foot trajectory and thus a continuous velocity trajectory. Figure 5.4 shows the output foot position of the cubic spline generation.

In the y-direction, the swing foot undergoes two cubic spline trajectories. Let h be the desired step height as specified by the user. Each of the two splines is executed in a time of $\tau/2$. The first starts at a position of $y = y_{foot}$ with a velocity of zero. In a period of half of the return time, the first trajectory completes with a position of $y = y_{foot} + h$, and a velocity of zero. The second trajectory is symmetric to the first, ending at a position of $y = y_{foot}$ with zero velocity.

In the x-direction, the swing foot undergoes three cubic spline trajectories. The first serves to stop the foot's x-velocity in a specific distance, the second performs the swing of the leg forward, and the third causes the foot to match velocities with the ground upon touchdown. The table below summarizes the initial and final conditions

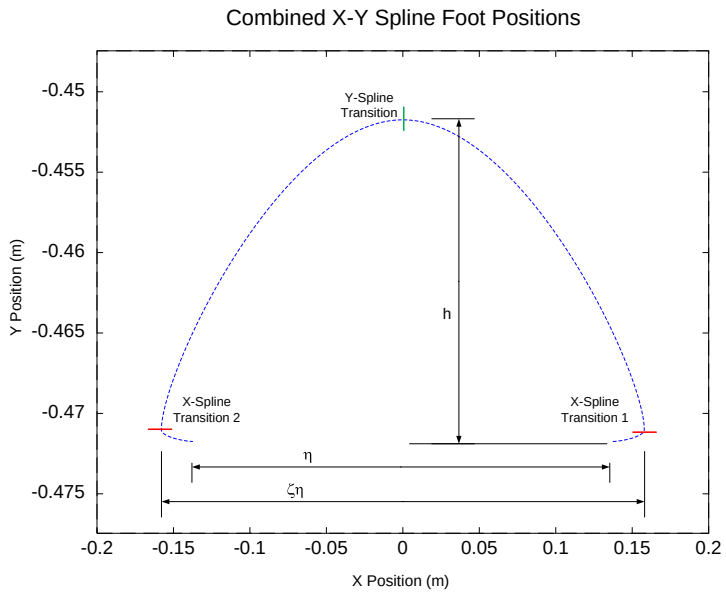

Figure 5.4: Five spline foot trajectory for swing leg

for each of the three splines. In these equations, ζ represents the horizontal width of the combined spline envelope normalized to the stroke. It is also shown in Fig. 5.4. The total time for each spline is directly proportional to the total displacement of each spline.

Spline	**Total Time**	x_0	$\dot{x}_0$	x_f	$\dot{x}_f$
Post-Liftoff	$\tau\frac{\zeta-1}{4\zeta-2}$	$\eta/2$	u	$\zeta\eta/2$	0
Swing	$\tau(1-2\frac{\zeta-1}{4\zeta-2})$	$\zeta\eta/2$	0	$-\zeta\eta/2$	0
Pre-Touchdown	$\tau\frac{\zeta-1}{4\zeta-2}$	$-\zeta\eta/2$	0	$-\eta/2$	u

Table 5.5: X-direction spline trajectories for the swing leg

Inverse Kinematics

Once the spline trajectories of the foot are used to calculate a desired foot position, these foot positions are passed to an inverse kinematics algorithm. This calculation produces the desired link angles needed to achieve the necessary foot locations. The derivation of the inverse kinematics can be found in section C.2.

5.4.2 Gravity Compensation

The gravity compensation strategy uses the current link angles, a desired weight support command, and a Jacobian transform to find the joint torques needed to maintain the current configuration. These desired torques are then converted into desired SEA deflections. This approach solves the inverse dynamics problem with the assumptions of massless links and a constant torso velocity. To produce a gravity balancing normal force, the end effector of the articulated leg (the foot), must exert a force on the ground that is equal to the gravitational force on the biped.

To produce this desired force, required joint torques are found using a Jacobian transpose matrix. Differentiating the forward kinematic results in section C.1, the Jacobian matrix is found. This is used to obtain a relation between the link angular velocities and the foot velocity, Eq. 5.6. It is also used to obtain a relation between the force provided by the end effector (the foot) and the torques required to produce that force, Eq. 5.7.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = l \begin{bmatrix} \cos\theta_t & \cos\theta_s \\ \sin\theta_t & \sin\theta_s \end{bmatrix} \begin{bmatrix} \dot{\theta}_t \\ \dot{\theta}_s \end{bmatrix} \tag{5.6}$$

$$\begin{bmatrix} \tau_h \\ \tau_k \end{bmatrix} = l \begin{bmatrix} \cos\theta_t & \sin\theta_t \\ \cos\theta_s & \sin\theta_s \end{bmatrix} \begin{bmatrix} f_x \\ f_y \end{bmatrix} \tag{5.7}$$

As described previously, the only force required is in the negative y-direction with a magnitude corresponding the the weight support commanded. From this observation combined with Eq. 5.7 the desired torques are as follows.

$$\begin{bmatrix} \tau_h \\ \tau_k \end{bmatrix} = -l \cdot W_{command} \begin{bmatrix} \sin\theta_t \\ \sin\theta_s \end{bmatrix} \tag{5.8}$$

If this torque is in the same direction as the SEAs possible deflection, the torques are divided by the spring constant to produce desired SEA deflections. If the torques are not in the same direction as the SEAs possible deflection, then the unidirectional hard stop will provide the necessary torques without the need for deflection.

5.5 *RobotBuilder* Simulation

Using all of the above relationships, the control strategy was implemented in *RobotBuilder*. The simulation demonstrated the need to account for the boom pitch as well as the circular walking path of the biped on the boom. To account for the circular path, the x-positions of each foot were multiplied by a ratio of the radius of the foot's motion over the radius of center of mass's motion.

To account for the boom pitch, pitch feedback was used to modify the inputs for the inverse kinematics. As the boom pitches, the coordinate system developed in Chapter 3, located at the right hip, also pitches. As this happens, the left hip gets closer to the ground than the right. This produces a problem, as the desired foot positions are ideally measured from an unrotated version of the coordinate frame. To correct for this, trigonometric relations are applied to the desired positions of each foot during the calculation of the inverse kinematics in order to place the foot at the correct location in the unrotated coordinate frame.

5.6 Implementation

To integrate the control strategy into the control framework discussed in Chapter 3, the walking controller had to be decomposed into a state based controller. Figure 5.5 shows the decomposition of the kinematic cycle into discrete states similar to as shown in Fig. 5.3. The INIT, POSITION, and HOLD states are added for physical implementation. The INIT state is used to set up all required variables and configurations on the Galil. The POSITION state places the legs in the starting configuration in a predetermined amount of time. The HOLD primitive then provides a period of inactivity for the experimenter to place the system on the ground. An audio cue is given to indicate the transition out of HOLD and into the kinematic cycle phase states. The kinematic cycle phase states are the same as discussed previously.

Due to the complexity of the non-linear feedback in the walking control strategy, the contour mode of motion was used to implement dynamically retargeted motor trajectories as dictated by the host. In this mode of motion, the user must supply motor trajectory "waypoints" for the Galil to track, and must ensure that the Galil is provided with waypoints at a sufficient frequency such that it never reaches the end of its waypoint buffer. Every millisecond, the motion planner and gravity compensation are executed on the host. The combined output of these calculations is then used to command a new motor waypoint at a resulting rate of 1kHz. These waypoints are commanded 2ms prior to being processed on the Galil, inherently introducing a 2ms feedback delay. This delay is acceptable, as it is very small in comparison to the large time constants that the system experiences during ground contact. In order to ensure reliable generation of waypoints every millisecond, the host was outfitted with a real-time Linux kernel. Prior to the inclusion of this kernel, the host would fail

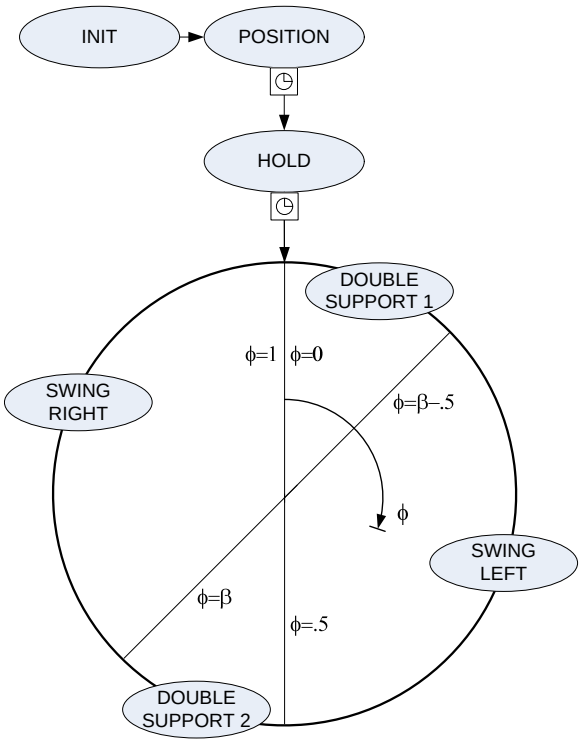

Figure 5.5: Walking state machine states

to generate waypoints at a sufficient frequency, causing the Galil buffer to become "cluttered" [35].

Physical implementation also demonstrated the need to account for the torso pitch. Although there is a pin which prevents any torso pitch, the fixed position of torso is about 3° forward of vertical with this pin in place. As a result, simple modifications were made to the inverse kinematics to account for this pitch.

5.7 Results

The walking controller has shown promising results in hardware for desired speeds ranging from .2 m/s to .45 m/s. Figure 5.6 shows the torso velocity over four strides

with a desired speed of .38 m/s. The speed is shown in its filtered and unfiltered form, with a fourth-order Butterworth filter used to smooth quantization errors in the unfiltered velocity. This filter uses a 2 Hz cutoff frequency. The actual performance

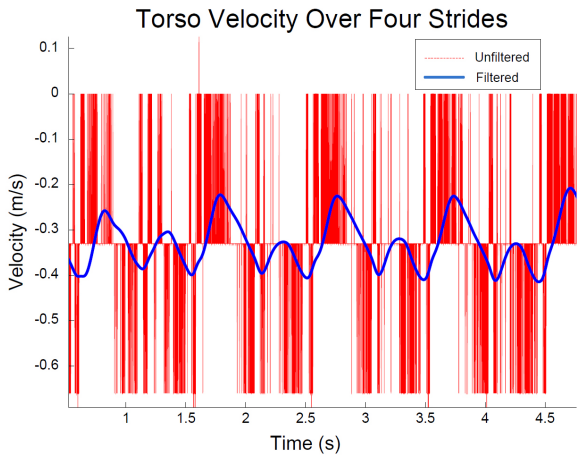

Figure 5.6: Torso velocity over four strides. Filtered velocity uses a fourth-order Butterworth with a cutoff frequency of 2 Hz.

of the controller results in a gait with an average speed of .33 m/s. The largest deviations from the desired velocity occur at impact, more specifically when impact occurs prior to desired impact. Figure 5.7 shows the torso height for the same four walking strides. This figure shows that the height of the hip if fairly well regulated by the gravity compensation strategies. The hip height is maintained within 1.5 cm of its desired position of 48.3 cm for this walk. Yet, there is a direct correlation between the undesired behaviors in each of these graphs. Upon a drop in hip height, signifying an incomplete weight compensation, the next footfall causes a decrease in speed. This weakness represents one of major areas of possible improvement for the control strategy. After this particular experiment, it was found that the unidirectional

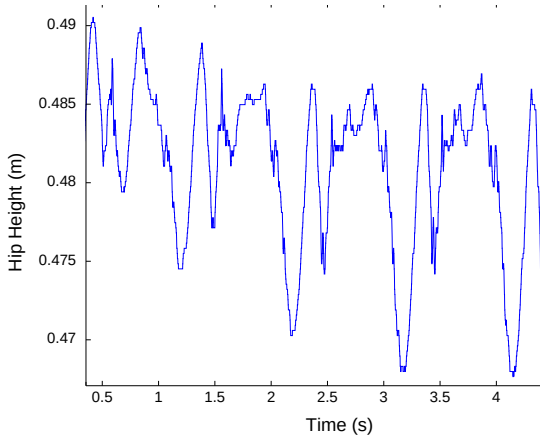

Figure 5.7: Hip height over four walking strides.

elastomer on the right knee had become worn through, resulting in a decreased pre-load on the right knee. While this demonstrates the controllers sensitivity to pre-load difference between the legs, it also highlights the approaches preliminary robustness, as the average speed still remained within 13% of the desired speed. While these results demonstrate the control approaches validity, they also present the opportunity for further investigations to refine the control strategy for more robust performance prior to its application to a full dynamic walk.

5.8 Summary

This chapter has presented the development and implementation of a walking control strategy. The layered structure of the control strategy has been emphasized, with details provided for the supervisory control and lower-level motion planning. Additionally, a gravity compensation strategy has been presented. The overall strategy has proven successful in simulation and through preliminary results in hardware.

CHAPTER 6

Summary and Conclusions

6.1 Summary and Conclusions

The results presented in this work demonstrate successful completion of each of the research objectives. First, a real-time distributed control system for the bipedal platform was effectively developed for use in dynamic movements. The modularity of the distributed control framework provides the experimenter with a simple way to extend any control strategy to hardware, and to understand its operation through real-time and post-control feedback. While the user and data structure interfaces developed are as simple as possible, the underlying complexity of the system should not be underestimated. The careful implementation efficient data structures and parallel operating threads allows the software to operate in real-time, yet is completely transparent to the high-level control strategies. As part of this distributed control system, the foundational control system's operation has provided effective real-time position control of the biped's actuators. The senor system developed, while once again hidden from the high-level programmer, has been carefully implemented with proper noise-limiting practices and has been augmented with signal conditioning circuitry.

The distributed control framework has been successfully applied to the implementation of both jumping and walking control. While investigation by Hester in

[20] produced a control strategy for jumping, many hardware implementation issues were addressed in this work. The state transitions have been shown to occur with satisfactory time delays which meet the stringent real-time deadlines for control of high-power motion.

In addition, a layered control strategy for walking has been developed and implemented in hardware. Through supervisory speed control, motion planning, gravity compensation, and foundational control, the performance of the system has shown the control approach's validity at a variety of speeds in hardware.

Overall, this research has provided a significant step towards the execution of multiple dynamic maneuvers in hardware. Future work should be able to build off of the control strategies already in place to create complex motions that demonstrate the full mobility of the biped.

6.2 Suggestions for Future Work

Control development for the experimental biped has only breached the surface of full investigation into dynamic maneuvers on this platform. The following suggestions outline possible areas for future research with this platform and with the execution of dynamic movements in general.

1. The biped's sensing systems currently provide adequate performance for the execution of a locked-torso jump and a walk. As movements become less and less constrained, they become more sensitive to inaccurate feedback. Additional modeling and state estimation techniques should be investigated to provide accurate feedback that cannot be directly deduced from the sensor systems. The real-time performance of these estimation techniques should be thoroughly

studied, as a high-fidelity estimation scheme may interfere with the strict time restrictions on the control.

2. While the walking results demonstrate the validity of the control approach for the locked-torso case, investigation into full dynamic walking with SEAs should take place. Major modifications will need to be made when the torso pitch DOF is used. Simulations have tested the removal of the locked-torso condition during the steady-state performance of a gait, which results in the the biped's loss of stability within two steps. The walking control strategy could also be improved through the study of: 1) the removal of the massless legs assumption. 2) the effects of the SEA pre-load, and its variability from leg to leg, on the success of gravity compensation. 3) estimation techniques for the inclusion of velocity feedback in the gravity compensation strategy.

3. Current plans involve the implementation of a run in hardware by Liu in the future. Once this work is complete, transitions between movements should be investigated. The transition from a walk to a run and back could provide insight into the system and aid the development of more rapid maneuvers such as a rapid stop. Ultimately, complex combinations of motions should be investigated to fully address the transient behaviors involved in non-periodic dynamic motion.

4. The system's current jumping controller was built to achieve precise repetitive jump heights. It was designed for maximum jump heights of 60 cm. Still, this height does not reach the maximum jump heights predicted during design in [17]. Genetic algorithms should be applied to optimize the jump heights for

this system in order to realize its full jump height potential. Additionally, the use of genetic algorithms for the realization of specific movement controllers should be investigated. While this approach could be used in simulation, the addition of machine learning should be studied in hardware to fine tune control strategies. The optimal balance between these two types of efforts could produce high-performance motion in hardware while minimizing the wear to the system.

It is clear that there are a number of exciting avenues of research that are yet to be investigated for this platform. As these research avenues are explored, their insights should be used to understand the essence of dynamic stability for the bipedal structure. Ultimately, these insights should be extended to three dimensions and used to develop control strategies for fully autonomous bipeds.

This thesis has presented the design of a control framework that can be extended to develop control strategies for a wide range of dynamic maneuvers for this platform. Hopefully this framework will prove fruitful for further experimental endeavors into the nature of dynamic stability and ultimately the realization of fully autonomous bipeds that can dynamically adapt to their environments.

APPENDIX A

System Parameters

System parameters for the biped's overall configuration, actuators, and actuator drives are included in the following sections. The description of the overall experimental setup begins on the next page.

A.1 Physical System Parameters

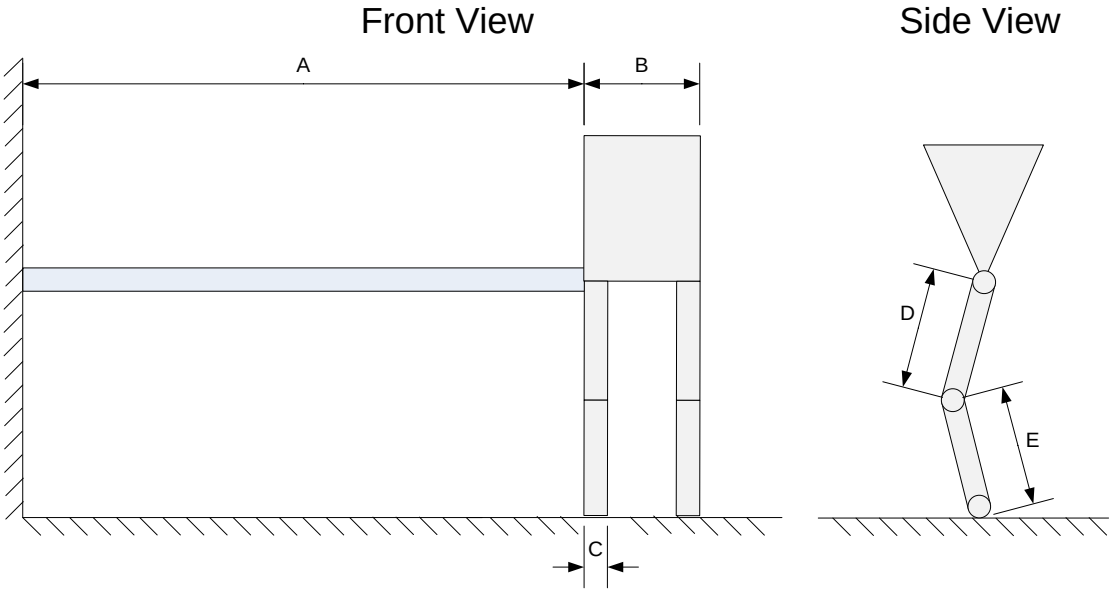

Figure A.1: Biped physical parameters

Letter	Parameter	Description	Value
A	Boom Length	From center of boom yaw axis to the outside of the right hip	2.05162 m
B	Torso Width	From the outside of the right hip to the outside of the left hip	.24765 m
C	Leg Width	Twice the distance from the outside of the right hip to the center of the thigh	.05677 m
D	Thigh Length	From hip axis to the knee axis	.25 m
E	Shank Length	From the knee axis to the bottom of the foot	.25 m

Table A.1: Numeric values of biped physical parameters

A.2 Actuator and Motor Drive Parameters

Parameter	Units	Value
Values at Nominal Voltage		
Nominal Voltage	V	48
No Load Speed	rpm	16500
No Load Current	mA	422
Nominal Speed	rpm	15800
Nominal Torque (max continuous)	mNm	120
Nominal Current (max continuous)	A	4.7
Stall Torque	mNm	3430
Starting Current	mA	124
Max Efficiency	%	89
Characteristics		
Terminal Resistance Phase to Phase	Ω	0.386
Terminal Inductance Phase to Phase	mH	0.0653
Torque Constant	mNm / A	27.6
Speed Constant	rpm / V	346
Speed / Torque Gradient	rpm / mNm	4.83
Mechanical Time Constant	ms	1.68
Rotor Inertia	gcm^2	33.3
Thermal Data		
Thermal resistance housing-ambient	K /W	5.3
Thermal resistance winding-housing	K /W	0.0785
Thermal time constant winding	s	0.738
Thermal time constant motor	s	848
Max. permissible winding temperature	°C	+155

Table A.2: Motor parameters for Maxon EC-powermax 30

Parameter	Value
Spring Constant	30 N/rad
Max Deflection	2.167 rad

Table A.3: SEA parameters

Parameter	Units	Value
DC Supply Voltage	VDC	16-80
Peak Current (2 sec. max., internally limited)	A	$\pm$ 12
Max. Continuous Current (internally limited)	A	$\pm$ 6
Minimum Load Inductance	μH	100
Switching Frequency	kHz	33 $\pm$ 15%
Heatsink Temperature Range	°C	0 to +75, disables if $>$ 75
Power Dissipation At Continuous Current	W	24
Over-Voltage Shut-Down	V	88

Table A.4: Amplifier parameters for AMC ZBDC12A8.

APPENDIX B

Analog Filter Approximations

B.1 Butterworth Approximation

The Butterworth filter is the most simple approximation to an ideal low-pass filter. The filter provides a maximally flat frequency response in the passband and monotonically increasing attenuation in the stopband. While [31] was used as a reference for this work, a treatment of this filter type can be found in countless textbooks. The n-th order analog Butterworth filter has all n zeros at infinity. Its pole locations are described below for a cutoff frequency ω_c.

$$p_k = \omega_c e^{\frac{j(2k+n-1)\pi}{2n}}, k = 1, 2, ..., n \tag{B.1}$$

B.2 Chebyshev Approximation

The Chebyshev (or Tschebyscheff) filter achieves increased stopband attenuation but sacrifices pass band ripple. The minimum stopband gain is qualified by the ripple factor ϵ, according to Eq. B.2. A larger acceptable ripple in the passband causes the filter to sharpen its transition to the stopband. Once again [31] was used as a reference for the analog filter pole and zero locations. The n-th order analog filter has

all n zeros at infinity. Its pole locations are described below in Eq. B.3 for a cutoff frequency ω_c and ripple factor ϵ.

$$G_{min} = \frac{1}{\sqrt{1+\epsilon^2}} \tag{B.2}$$

$$\begin{aligned} p_k &= -\omega_c \sinh\left(\frac{1}{n}\text{arsinh}\left(\frac{1}{\varepsilon}\right)\right)\sin\left(\frac{\pi}{2}\frac{2k-1}{n}\right) \\ &\quad +j\omega_c \cosh\left(\frac{1}{n}\text{arsinh}\left(\frac{1}{\varepsilon}\right)\right)\cos\left(\frac{\pi}{2}\frac{2k-1}{n}\right), k = 1, 2, ..., n \end{aligned} \tag{B.3}$$

B.3 Elliptic Approximation

The Elliptic approximation improves upon the Chebyshev through the allowance of ripple in the passband. The ripple factor, which defines the minimum stopband gain (Eq. B.2) is once again used. In addition, a quantity known as the discrimination factor, L_n, is used with the ripple factor to qualify the maximum stopband gain as described in Eq. B.4. While an elegant implementation of this filter type may have been possible through a complete understanding of the elliptic rational functions that are used to create the elliptic approximation, a number of series approximations were used to quickly implement the analog elliptic poles and zeros.

$$G_{max} = \frac{1}{\sqrt{1+L_n^2\epsilon^2}} \tag{B.4}$$

Before proceeding, a few useful functions will be defined as described in [36]. The complete elliptic integral of the first kind, $K(m)$, is defined in Eq. B.5. This function is approximated using trapezoidal integration in code. The elliptic nome, $q(m)$, is defined in Eq. B.6 with the use of the complete elliptic integral of the first kind. Finally, the Pochhammer symbol is introduced in Eq. B.7.

$$K(m) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1-m\sin^2\phi}} \tag{B.5}$$

$$q(m) = e^{-\frac{\pi K(1-m)}{K(m)}} \tag{B.6}$$

$$(x)_n = \prod_{k=0}^{n-1}(x+k) \tag{B.7}$$

The rest of the section will provide a reference of the equations implemented in the code without regards to their theoretical backgrounds. The selectivity factor of the filter, ξ, is found by the order-equation as detailed in [37]. The code uses a recursive interval halving algorithm to approximate a solution, ξ, to Eq. B.8.

$$n\frac{K\left(\frac{1}{L_n^2}\right)}{K\left(1-\frac{1}{L_n^2}\right)} = \frac{K\left(\frac{1}{\xi^2}\right)}{K\left(1-\frac{1}{\xi^2}\right)} \tag{B.8}$$

The pole-zero formulas will use the selectivity factor as well as the Jacobi elliptic cd function and its inverse, Eqs. B.10 and B.9 respectively. They are defined using series representations from [36]. It should be noted that the cosine and inverse cosine functions in these formulas are the more general, complex versions of these functions.

$$\mathrm{cd}^{-1}(z,m) = \sum_{k=0}^{\infty}\frac{\left(\frac{1}{2}\right)_k^2}{(k!)^2}\left(\cos^{-1}(z) + \frac{\sqrt{1-z^2}}{2z}\sum_{j=1}^{k}\frac{(j-1)!z^{2j}}{\left(\frac{1}{2}\right)_j}\right)m^k \tag{B.9}$$

$$\mathrm{cd}(z,m) = \frac{2\pi}{\sqrt{m}K(m)}\sum_{k=0}^{\infty}\frac{(-1)^k q(m)^{k+1/2}}{1-q(m)^{2k+1}}\cos\left(\frac{(2k+1)\pi z}{2K(m)}\right) \tag{B.10}$$

The poles and zeros of the filter are finally found using formulas from [38].

$$p_m = j\omega_c\,\mathrm{cd}\left[\frac{K(1/\xi^2)}{nK(1/L_n^2)}\mathrm{cd}^{-1}\left(\frac{j}{\epsilon},\frac{1}{L_n^2}\right) + \frac{2mK(1/\xi^2)}{n}, 1/\xi^2\right] \tag{B.11}$$

$$z_m = \omega_c\frac{\xi}{j\,\mathrm{cd}\left(K(1/\xi^2)\,\frac{4m-1}{n},\frac{1}{\xi^2}\right)}, m = 0,1,...,n-1 \tag{B.12}$$

The results of these equations were verified using filter design tools in Matlab. While a hardly elegant implementation, the resulting filters are the lowest order filters needed to meet frequency response specifications. Although they were not used heavily in the system, these filters would save processing time during use due to their improved order over other types of filters.

APPENDIX C

Kinematics Calculations

C.1 Forward Kinematics

Forward kinematics involves finding the position of the biped's foot as a function of its joint angles. In this case, the forward kinematics for the position of the foot can be derived through the application of a few simple trigonometric relationships. In Fig. C.1 below, the coordinate system is rigidly attached to the torso, with the origin coinciding to the center of the hip axis.

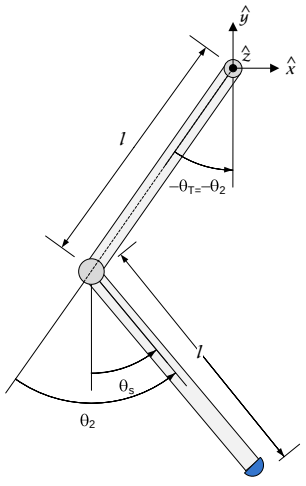

Figure C.1: Kinematic model of leg

To aid in the solution of the inverse kinematics in the following section, define θ_1 and θ_2 as follows:

$$\theta_1 = \theta_t \tag{C.1}$$
$$\theta_2 = \theta_s - \theta_1 \tag{C.2}$$

By inspection, the coordinates of the location of the knee joint are:

$$x_k = l \sin \theta_t \tag{C.3}$$
$$= l \sin \theta_1 \tag{C.4}$$

$$y_k = -l \cos \theta_t \tag{C.5}$$
$$= -l \cos \theta_1 \tag{C.6}$$

Similarly, the coordinates of the location of the foot are:

$$x_f = l \sin \theta_t + l \sin \theta_s \tag{C.7}$$
$$= l \sin \theta_1 + l \sin(\theta_1 + \theta_2) \tag{C.8}$$

$$y_f = -l \cos \theta_t - l \cos \theta_s \tag{C.9}$$
$$= -l \cos \theta_1 - l \cos(\theta_1 + \theta_2) \tag{C.10}$$

C.2 Inverse Kinematics

Inverse kinematics involves solving a set of trigonometric functions to find sets of joint angles that would produce a desired location of the foot relative to the torso. The approach used is borrowed from [39]. By squaring Equations (C.8) and (C.10) and adding their results the following is obtained:

$$2l^2 + 2l^2 \left[s_1 s_{12} + c_1 c_{12} \right] = x_f^2 + y_f^2 \tag{C.11}$$

$$\cos \theta_2 = \frac{x_f^2 + y_f^2 - 2l^2}{2l^2} \tag{C.12}$$

$$\theta_2 = \cos^{-1}\left[\frac{x_f^2 + y_f^2 - 2l^2}{2l^2}\right] \tag{C.13}$$

$$= \text{atan2}\left(\sqrt{4l^4 - (x_f^2 + y_f^2 - 2l^2)^2}, x_f^2 + y_f^2 - 2l^2\right) \tag{C.14}$$

There should be two unique physical solutions to the inverse kinematics problem, represented by the two equal and opposite values attained by Equation (C.13) in the interval $[-\pi, \pi]$. Yet, the only solution that will lie within the working range of the leg will have $\theta_2 > 0$, thus only the positive value of θ_2 is used. The thigh angle can be found through the combination of this result with the expansion of Equation (C.8).

$$x_f = ls_1 + l(s_1c_2 - c_1s_2) \tag{C.15}$$

$$x_f = (l + lc_2)s_1 - (ls_2)c_1 \tag{C.16}$$

Now define two new variables, a and b, as functions of θ_2 as follows:

$$a = -l\sin\theta_2 \tag{C.17}$$

$$b = l + l\cos\theta_2 \tag{C.18}$$

Finally, using the previously defined variables, the required thigh angle is found as follows:

$$\theta_1 = \text{atan2}(b, a) - \text{atan2}(\sqrt{a^2 + b^2 - x_f^2}, x_f) \tag{C.19}$$

APPENDIX D

Biped Wiring

D.1 Power System

This section outlines the circuitry to provide power to the amplifiers and motion controller. A table of all required parts and their corresponding part numbers is included in Table D.3.3 on page 100.

D.1.1 Motion Controller Power

The motion controller is powered by a 48V power supply (P5). The supply is hooked up to a standard 120VAC outlet with a 3-conductor cable (P5), and 3 tongue spades (P7). A power and ground line (P7) are used to provide power across the boom to the motion controller. A crimp connection provided by Galil is finally used to connect the power and ground to the motion controller.

D.1.2 Amplifier Power

The amplifiers are powered by an array of four 48V power supplies (P8) which operate in parallel. The supplies are hooked up to a high-power 240VAC outlet. First, a NEMA 15-30 connector (P3) interfaces with the outlet. A 3-conductor cable (P2) runs the ground and 2 of the AC phases to a female connector (P4) near the

power supply array. A male connector (P3) interfaces with this female connector, and uses the same 3-conductor cable to provide power to each supply. The ground and AC phases interface with the power supply using tongue spades (P13). Parallel operation circuitry and remote on/off sensing are described in Sec. 2.7.1. Crimps and crimp housings for these connections came with the supplies. The 48V output of each supply and the ground of each supply is wired to a separate terminal block (P9) which has each row connected in parallel with a jumper (P10). The blocking diode circuitry (P15, P14) is wired to these terminal blocks as described in Sec. 2.7.2. The output of the blocking diode circuitry is sent across the boom to an identical set of terminal blocks using 10AWG wire (P16) and a set of ring spades (P11). A power and ground line for each amplifier interface with these blocks using tongue spades (P17) and are run to the amplifiers with 18 AWG wire (P7). Each wire is crimped into a butt splice (P18) which also houses a set of 22 AWG wires (P19). Each of these wires is crimped (P20) and inserted into a housing (P21). The crimp housing mates with the amplifier on amplifier pins 2.1-2.5. The crimps are placed so that when mated with the amplifier the ground lines are tied to P2.2 and P2.3, while the power lines are tied to P2.4 and P2.5.

D.2 Connector Pinout Reference

This section provides a quick reference for the various connections for the system. Additional pinouts can be found in each product's data sheet.

Pin#	Label	Description	Pin#	Label	Description	Pin#	Label	Description
1	ERR	Error Output	16	RST	Reset Input	31	GND	Digital Ground
2	DI1	Digital Input 1/ A latch	17	INCOM	Input Common	32	DI2	Digital Input 2 / B latch
3	DI4	Digital Input 4 / D latch	18	DI3	Digital Input 3 / C latch	33	DI5	Digital Input 5
4	DI7	Digital Input 7	19	DI6	Digital Input 6	34	DI8	Digital Input 8
5	ELO	Electronic Lock Out	20	ABRT	Abort Input	35	GND	Digital Ground
6	LSCOM	Limit Switch Common	21	N/C	No Connect	36	FLSA	Forward Limit Switch A
7	HOMA	Home Switch A	22	RLSA	Reverse Limit Switch A	37	FLSB	Forward Limit Switch B
8	HOMB	Home Switch B	23	RLSB	Reverse Limit Switch B	38	FLSC	Forward Limit Switch C
9	HOMC	Home Switch C	24	RLSC	Reverse Limit Switch C	39	FLSD	Forward Limit Switch D
10	HOMD	Home Switch D	25	RLSD	Reverse Limit Switch D	40	GND	Digital Ground
11	OPWR	Output Power	26	N/C	No Connect	41	DO1	Digital Output 1
12	DO3	Digital Output 3	27	DO2	Digital Output 2	42	DO4	Digital Output 4
13	DO6	Digital Output 6	28	DO5	Digital Output 5	43	DO7	Digital Output 7
14	ORET	Output Return	29	DO8	Digital Output 8	44	CMP	Output Compare
15	+5V	+5V	30	+5V	+5V			

Figure D.1: Galil digital I/O connector pinout [2]

Pin #	Label	Description
1	AGND	Analog Ground
2	AI1	Analog Input 1
3	AI3	Analog Input 3
4	AI5	Analog Input 5
5	AI7	Analog Input 7
6	AGND	Analog Ground
7	-12V	-12V from Controller
8	+5V	+5V from Controller
9	AGND	Analog Ground
10	AI2	Analog Input 2
11	AI4	Analog Input 4
12	AI6	Analog Input 6
13	AI8	Analog Input 8
14	N/C	No Connect
15	+12V	+12V from Controller

Figure D.2: Galil analog input connector pinout [2]

Pin #	Label	Description	Pin #	Label	Description
1	RES	Reserved [1]	14	FLS	Forward Limit Switch Input
2	AEN	Amplifier Enable	15	AB+	B+ Aux Encoder Input
3	DIR	Direction	16	MI-	I- Index Pulse Input
4	HOM	Home	17	MB+	B+ Main Encoder Input
5	LSCOM	Limit Switch Common	18	GND	Digital Ground
6	AA-	A- Aux Encoder Input	19	MCMD	Motor Command
7	MI+	I+ Index Pulse Input	20	ENBL+	Amp Enable Power
8	MA-	A- Main Encoder Input	21	RES	Reserved [3]
9	+5V	+5V From Controller	22	RLS	Reverse Limit Switch Input
10	GND	Digital Ground	23	AB-	B- Aux Encoder Input
11	ENBL-	Amp Enable Return	24	AA+	A+ Aux Encoder Input
12	RES	Reserved [2]	25	MB-	B- Main Encoder Input
13	STP	PWM/Step	26	MA+	A+ Main Encoder Input

Figure D.3: Galil axis connector pinout [2]

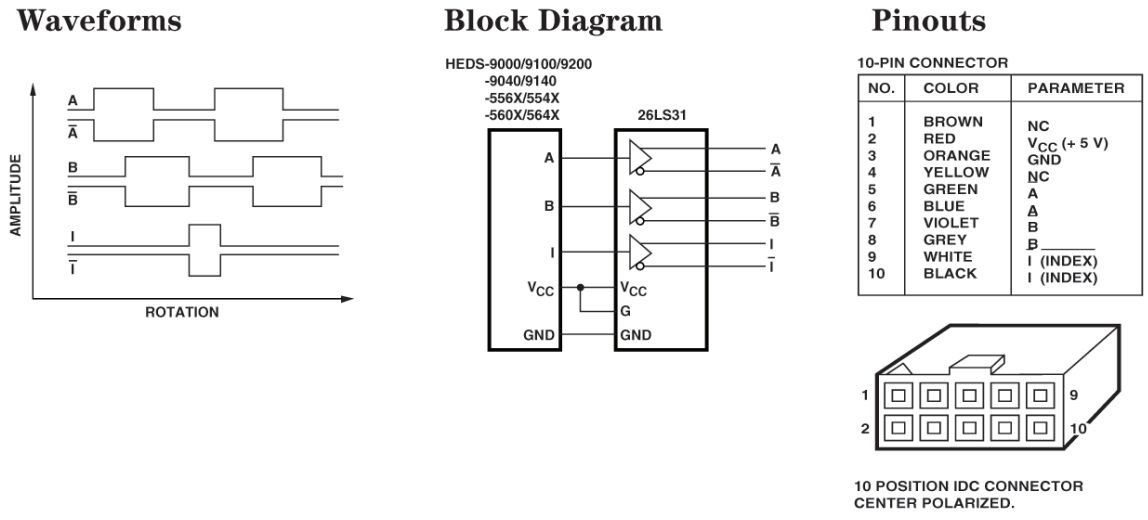

Figure D.4: Motor encoder connections [3]

CONNECTOR	PIN	NAME	DESCRIPTION / NOTES	I/O
P1	1	PWM IN	10 – 25 kHz pulse width modulated digital input command (+5V). Input duty cycle commands the output current.	I
	2	SIGNAL GROUND	Reference ground	GND
	3	DIR	Direction input	I
	4	CURRENT MONITOR OUT	Output voltage proportional to motor output current: ZBDC6A6: 1V = 2A; ZBDC12A8: 1V = 4A	O
	5	INHIBIT IN	This TTL level input signal turns off all power devices of the "H" bridge when pulled to ground (when JE1 is installed), which is a fault condition. If the JE1 jumper is removed, pulling this pin to ground will enable the outputs.	I
	6	+V HALL OUT	+6VDC @ 30 mA output for Hall sensor power	O
	7	SIGNAL GROUND	Reference ground	GND
	8	HALL 1	Hall sensor inputs; TTL logic levels; internal 5kΩ pull-up to 5V. The standard commutation is for 120-degree phased motors. For 60-degree motors, JE2 must be removed.	I
	9	HALL 2		
	10	HALL 3		
	11	CURRENT REF OUT	Monitors the input signal connected directly to the internal current amplifier. 7.25V = max. peak current.	O
	12	FAULT OUT	TTL level output. Becomes high during output short circuit, over-voltage, over temperature and power-up reset.	O
P2	1	RESERVED	Reserved	
	2	POWER GROUND	Power ground (current rating per pin = 3A)	GND
	3			
	4	HIGH VOLTAGE	DC Power Input (current rating per pin = 3A)	I
	5			
	6	NC	(no connection; pin removed)	
	7	MOTOR C	Motor phase C connection (current rating per pin = 3A)	O
	8			
	9	MOTOR B	Motor phase B connection (current rating per pin = 3A)	O
	10			
	11	MOTOR A	Motor phase A connection (current rating per pin = 3A)	O
	12			

Figure D.5: Connections for the AMC ZBDC12A8 [22]

Wire	Color
Motor Connection (Cable AWG 18)	
Motor winding 1	red
Motor winding 2	black
Motor winding 3	white
Sensor Connection (Cable AWG 26)	
Hall sensor 1	red/grey
Hall sensor 2	black/grey
Hall sensor 3	white/grey
GND	blue
VHall (4.5 - 24 VDC Input)	green

Table D.1: Maxon EC-powermax 30 connections [21]

D.3 Sensor and Motor Connections

The following sections will define the sensor and motor connections. A table of all required parts and their corresponding part numbers is included in Table D.3.3 on page 101. Table D.2 below describes which axis each actuator uses on the Galil.

Actuator	Axis
Right Knee	A
Right Hip	B
Left Hip	C
Left Knee	D

Table D.2: Motor axis designations

D.3.1 Amplifier Connections

Amplifier to Galil

Pins P1.1 through P1.12 are low-power connections to the amplifier. They interface with hall effect sensors and the Galil motion controller. This connection uses a 12-position crimp housing (S3) along with a number of crimps (S4). All further connections to the amplifier in this and the next section will use this crimp connection to the amplifier. Each amplifier is connected to the Galil through a D-Sub axis connector (S1, S7). Ribbon cable is soldered to the D-Sub connector as described in Table D.3, crimped, and inserted into the necessary housing.

Signal	Wire Color	Amp Pin	Galil Pin
PWM	white	P1.1	13
Ground	gray	P1.2	10
Direction	purple	P1.3	3
Inhibit	blue	P1.5	2

Table D.3: Amplifier to Galil connections

Amplifier to Hall Effect Sensors

Each amplifier is connected to its motor's hall effect sensors through ribbon cable and a locking connector. The wires crimped at the amplifier crimp housing are first connected to a female crimp connector (S14, S15). The mating male connector (S12, S13) is directly connected to the hall effect sensor wires. Table D.4 describes this connection in more detail.

Signal	Amp Wire Color	Amp Pin	Hall Wire Color
Hall sensor 1	orange	P1.8	red/grey
Hall sensor 2	yellow	P1.9	black/grey
Hall sensor 3	green	P1.10	white/grey
GND	red	P1.7	blue
VHall	brown	P1.6	green

Table D.4: Amplifier to hall effect sensor connections

Amplifier to Motor Connections

Pins P2.7 through P2.12 are high-power connections to the amplifier that interface with the motor phases. This connection uses a 6-position crimp housing (S13) along with a number of crimps (S4). These pins are connected to the motor phases as shown in Table D.5. As prescribed by the table, each set of two amplifier wires (S8) is crimped into a butt splice (S6) which is also crimped to the motor phase.

Name	Amp Wire Color	Amp Pins	Motor Wire Color
Winding 1	red	P2.11 & P2.12	red
Winding 2	black	P2.9 & P2.10	black
Winding 3	green	P2.7 & P2.8	white

Table D.5: Amplifier to motor phase connections

D.3.2 Encoder to Galil Connections

Each motor also has an encoder which is interfaced to its corresponding Galil axis D-Sub. The female connector provided at the end of the encoder is first connected

to an IDC Cable (S5). The wires from the IDC cable are soldered into the D-Sub connector as shown in Table D.6

Signal	**IDC Pin**	**IDC Wire Color**	**Galil Pin**
VCC	2	tan	9
GND	3	red	18
$\bar{A}$	5	orange	8
A	6	tan	26
$\bar{B}$	7	yellow	25
B	8	tan	17
$\bar{I}$	9	green	16
I	10	tan	7

Table D.6: Encoder to Galil connections

D.3.3 Boom/Torso Encoder to Galil

Each boom and torso encoder is interfaced with a Galil auxiliary encoder input on a particular axis. Table D.7 outlines the axis designations for these connections.

Actuator	**Axis**
Boom Yaw	A
Boom Pitch	B
Torso Pitch	C

Table D.7: Boom/torso encoder axis designations

Boom Encoders to Galil

The two boom encoders' signals are passed via shielded twisted pair cable (S11) to a male D-Sub connector (S18, S7). This connector then interfaces with a female D-Sub (S12) which is rigidly attached to the electronics board. Table D.8 details these connections.

Signal	D-Sub Pin	Galil Pin
Yaw A	7	A 24
Yaw $\bar{A}$	8	A 6
Yaw B	6	A 15
Yaw $\bar{B}$	5	A 23
Yaw GND	13	A 10
Yaw +5V	14	A 9
Yaw Shield	15	A 10
Pitch A	2	B 24
Pitch $\bar{A}$	1	B 6
Pitch B	3	B 15
Pitch $\bar{B}$	4	B 23
Pitch GND	11	B 10
Pitch +5V	10	B 9
Pitch Shield	9	B 10

Table D.8: Boom encoder to Galil connections

Torso Encoder to Galil

The torso encoder is connected through ribbon cable and a locking connector. The wires from the encoder are first connected to a female crimp connector (S14, S15). The mating male connector (S12, S13) is connected to ribbon cable and then to the Galil C axis D-Sub. Table D.9 describes this connection in more detail.

Signal	Encoder Wire Color	Ribbon Cable Color	Galil Pin
A	brown	brown	C 24
B	blue	white	C 15
GND	black	black	C 10
+5V	red	red	C 9

Table D.9: Torso encoder to Galil connections

Part Num.	Name	Manufacturer	Man. Part Num.	Vendor	Vendor Part Num.	Quantity
P1	AC Plug NEMA 15-30 Male	Hubbell	HBL8431C	StayOnline	6663	1
P2	3-Conductor Wire (12 AWG)	-	-	Graybar	-	40 ft
P3	AC Plug NEMA 6-20 Male	Hubbell	HBL5466C	StayOnline	6247	1
P4	AC Connector NEMA 6-20 Female	Hubbell	HBL5469C	StayOnline	6188	1
P5	48VDC Power Supply	Galil	PSR-6-48	Galil	PSR-6-48	1
P6	Power Cord	Qualtek	211012-06	Digi-Key	Q122-ND	1
P7	2-Conductor Wire (18 AWG)	RadioShack	278-567	RadioShack	278-567	20 ft
P8	48VDC Power Supply	Lambda	SWS600L-48	Digi-Key	285-1791-ND	4
P9	Barrier Block	Cinch	4-142	Digi-Key	CBB304-ND	4
P10	Terminal Jumper	Cinch	95B	Digi-Key	CBB318-ND	12
P11	Ring Spade #8 Stud	Tyco Electronics	2-320568-3	Digi-Key	A27357-ND	4
P12	Tongue Spade #8 Stud	Molex	19144-0039	Digi-Key	WM18360-ND	30
P13	Tongue Spade #10 Stud (10-12 AWG)	Molex	19144-0042	Digi-Key	WM18361-ND	10
P14	Capacitor 10000μF	Panasonic	ECE-T2AP103EA	Digi-Key	P10038-ND	1
P15	Schottky Blocking Diode	Microsemi	SBR8050	Digi-Key	SBR8050-ND	1
P16	10-Gauge Wire	RadioShack	278-569	RadioShack	278-569	35 ft
P17	Tongue Space #8 Stud (18-22 AWG)	Molex	19144-0003	Digi-Key	WM18353-ND	20
P18	Butt Splice	Molex	19154-0004	Digi-Key	WM18383-ND	20
P19	Hookup Wire (3 Color, 22 AWG)	RadioShack	278-1224	RadioShack	278-1224	10 ft
P20	Crimps	Molex	08-50-0113	Digi-Key	WM1114CT-ND	100
P21	Crimp Housing (5 Position)	Molex	22-01-3057	Digi-Key	WM2003-ND	4

Table D.10: Power system parts list

Part Num.	Name	Manufacturer	Man. Part Num.	Vendor	Vendor Part Num.	Quantity
S1	26-Pin HD D-Sub (Axis)	Norcomp	180-026-103L001	Digi-Key	T826ME-ND	4
S2	Ribbon Cable	-	-	-	-	-
S3	Crimp Housing (12 Position)	Molex	22-01-3127	Digi-Key	WM2010-ND	4
S4	Crimps	Molex	08-50-0113	Digi-Key	WM1114CT-ND	100
S5	IDC Cable	3M	M1RXK-1040K	Digi-Key	M1RXK-1040K-ND	4
S6	Butt Splice	Molex	19154-0004	Digi-Key	WM18383-ND	20
S7	D-Sub Backshell (Axis)	Norcomp	970-015-010R011	Digi-Key	970-15BPE-ND	5
S8	Hookup Wire (3 Color, 22 AWG)	RadioShack	278-1224	RadioShack	278-1224	10 ft
S9	44-Pin HD D-Sub	Norcomp	180-044-103L001	Digi-Key	T844ME-ND	1
S10	D-Sub Backshell (Dig. I/O)	Norcomp	970-025-010R011	Digi-Key	970-25BPE-ND	1
S11	4-Cond Shielded Twisted-Pair Cable	-	-	-	-	20 ft
S12	15-Pin SD D-Sub (Boom Enc.)	Norcomp	171-015-203L031	Digi-Key	3215FE-ND	1
S13	Crimp Housing (6 Position)	Molex	22-01-3067	Digi-Key	WM2004-ND	4
S14	Female Crimp Housing w/Latch	Molex	50-57-9405	Digi-Key	WM2903-ND	5
S15	Female Crimp	Molex	16-02-1112	Digi-Key	WM2571CT-ND	25
S16	Male Crimp Housing w/Slot	Molex	70107-0004	Digi-Key	WM2536-ND	25
S17	Male Crimp	Molex	16-02-0105	Digi-Key	WM2565CT-ND	25
S18	15-Pin SD D-Sub	Norcomp	171-015-103L001	Digi-Key	215ME-ND	1

Table D.11: Sensor parts list (not including parts for signal conditioning circuitry)

APPENDIX E

Detailed Galil Configuration

The following pages provide a detailed configuration for the Galil Motion Controller. The subroutines in this source provide support to the safety routines and state based controllers. All the startup commands for the Galil are included in the #AUTO routine. The first 15 lines of this section are detailed below to explain the Galil's core configuration. The rest of the routine is used to initialize variables for other subroutines.

```
#AUTO
'Configures the encoders as quadrature and defines their polarity
CE0,8,10,2;

'Configures the motors as PWM/Direction drive and defines their polarity
MT -1.5,-1.5,1.5,1.5

'Sets the torque limits to the maximum
TL9.9999999999,9.9999999999,9.9999999999,9.9999999999

KD886,886,886,886        'Sets the proportional gains for control
KP186,186,186,186        'Sets the derivative gains for control
ER-1,-1,-1,-1            'Disabled PD error limits
OE0,0,0,0                'Disables Off On Error Behavior
AQ1,3;AQ2,3;AQ3,3;AQ4,3; 'Configures the analog inputs as 0-5V inputs
CAS;VMCD;
CAT;VMBA;
hipFwLim=45900
hipBwLim=-45900
knFwLim=85102
knBwLim=-700
kp=186;
kd=886;
spA=500000;spB=0;spC=0;spD=0;
acA=1000000;acB=0;acC=0;acD=0;
dcA=1000000;dcB=0;dcC=0;dcD=0;
paA=0;paB=0;paC=0;paD=0;
retReq=0;
dH=.45;
```

```
30 EN
31
32 #ANALG
33 ZAX=@AN[5]*400;
34 WT2;
35 JP #ANALG
36 EN
37
38 #SAFETY
39 unsafe=0
40 rKnee=_TPA-_TPB
41 rHip=_TPB
42 lHip=_TPC
43 lKnee=_TPD - _TPC
44 IF (rKnee>knFwLim) | (rKnee<knBwLim)
45 MG "RKNEE" {EC}
46 unsafe=1
47 ENDIF
48 IF (lKnee>knFwLim) | (lKnee<knBwLim)
49 MG "LKNEE" {EC}
50 unsafe=1
51 ENDIF
52 IF (rHip>hipFwLim) | (rHip<hipBwLim)
53 MG "RHIP" {EC}
54 unsafe=1
55 ENDIF
56 IF (lHip>hipFwLim) | (lHip<hipBwLim)
57 MG "LHIP" {EC}
58 unsafe=1
59 ENDIFMG
60 IF unsafe=1
61 MG "UNSAFE" {EC}
62 MG "ER-ENC LIMITIS" {EC}
63 AB1
64 JS #HOME
65 ENDIF
66 JP #SAFETY
67 EN
68
69 #INITPD
70 OF0,0,0,0
71 KP 186,186,186,186
72 KD 886,886,886,886;SH;
73 EN
74
75 #RET
76 JP #RET,retReq=0
77 'MG "RET-ACK",TIME%65536 {EC}
78 retReq=0;t=.60*@SQR[(dH-.3)/4.9];t1=t/3;t2=t1*2;f=1/((.5*t1)+(.5*t2));
79 OE1,1,1,1;AB1;OE0,0,0,0
80 SP @ABS[knee-_TPA]*f,@ABS[hip-_TPB]*f,@ABS[hip-_TPC]*f,@ABS[knee-_TPD]*f
81 DC _SPA/t2,_SPB/t2,_SPC/t2,_SPD/t2
82 AC _DCA*2,_DCB*2,_DCC*2,_DCD*2
83 OF0,0,0,0;KP 186,186,186,186;KD 886,886,886,886
84 SH;PA knee,hip,hip,knee;BG;
85 MG "RET-COMP",TIME%65536 {EC}
86 EN
87
88 #HOME
89 VA2000000,2000000
90 VD2000000,2000000
91 VS5600,5600;SH;
92 CSS;
93 CST;
94 CAS;VP-21000-_TPC,21000-_TPD;VE;
```

```
CAT;VP-21000-_TPB,21000-_TPA;VE;
BGST
MC
EN

#PRCYCLE
MO
WT 3000
SH
SP9800,9800,9800,9800
PA0,0,0,0
BG
AM
CAS
VM CD
CAT
VM BA
CSS
CST
CAS
VS600000,600000
VA2000000,2000000
VD2000000,2000000
VP42000,105000;VE;
CAT
VP-42000,-21000;VE;
BGST
MC
x=0
#VP1LP
CAS
VP-42000-42000,-21000-105000;VE
CAT
VP42000+42000,21000+105000;VE
BGST
MC
CAS
VP42000+42000,21000+105000;VE
CAT
VP-42000-42000,-21000-105000;VE
BGST
MC
x=x+1
JP#VP1LP,x<20
EN

#STMOTS
AB1
HX 3
JS #INITPD
IF _MOA=0
IPA=-1*_TEA
ENDIF
IF _MOB=0
IPB=-1*_TEB
ENDIF
IF _MOC=0
IPC=-1*_TEC
ENDIF
IF _MOD=0
IPD=-1*_TED
ENDIF
MC
MO
EN
```

APPENDIX F

Biped Host Software

This chapter contains the most important source files for the distributed control software. It includes files which deal with the data thread, files for digital filters, and finally files for the state machines. While by no means a full coverage of the full software developed, these files include many of the important details discussed in the text.

F.1 Data Thread Files

DataThread.h

```
1 #ifndef DATATHREAD_H_
2 #define DATATHREAD_H_
3
4 #include <QThread>
5 #include "ReadingDefines.h"
6 #include "DataVector.h"
7
8 class DiscreteFilter;
9 class FFTData;
10 class QwtData;
11
12 class DataThread : public QThread
13 {
14     Q_OBJECT
15
16   enum CMType
17   {
18     ABSOLUTE,
19     DERIVATIVE
20   };
21
22   typedef struct CMInfoStruct
23   {
```

```
24      CMInfoStruct(ContMeasmnt c,Reading d,CMType t) : cReadingNum(c),dReadingNum(d),
            type(t) {};
25      CMInfoStruct() : cReadingNum(CM_SIZE),dReadingNum(NUM_READINGS),type(DERIVATIVE)
            {};
26      ContMeasmnt cReadingNum;
27      Reading dReadingNum;
28      CMType type;
29    } CMInfo;
30
31  public slots:
32    void restart();
33    void pause();
34      void resume();
35
36  public:
37      DataThread();
38      void stop();
39      void addLog(const string &s);
40      void saveLog();
41      static void initializeFilterData();
42      double systemTime();
43      QwtData& getFilterInput(int);
44      QwtData& getFilterOutput(int);
45      void runFilterFFTs(int);
46    DataDeque dataRecords;
47    DataRecord * currentData;
48    static vector<string> filterStrings,bandWidthStrings,orderStrings,pbRipStrings,
          sbRipStrings,typeStrings;
49  protected:
50      void run();
51
52  private:
53
54      volatile bool stopped;
55      volatile int updatePeriod;
56      volatile bool paused;
57
58      static vector<CMInfo> cmDataSet;
59
60      vector<DiscreteFilter *> filters;
61      vector<FFTData*> inputData;
62      vector<FFTData*> outputData;
63
64      vector<string> logMessages;
65      vector<double> logTimes;
66
67      volatile unsigned long secBase;
68      volatile unsigned long microSecBase;
69  };
70
71
72  #endif /*DATATHREAD_H_*/
```

DataThread.cpp

```
1  #include "DataThread.h"
2  #include "DataVector.h"
3  #include <QtGui>
4  #include <string>
5  #include <iostream>
6  #include <cstdio>
7  #include <stdlib.h>
```

```
 8 #include <sys/time.h>
 9 #include "GlobalVariables.h"
10 #include "GlobalDefines.h"
11 //#include "../../ThermalModel/src/h/ThreadCom.h"
12 #include  "FFTData.h"
13 #include "DiscreteButterworth.h"
14 #include "DiscreteChebychev.h"
15 #include "DiscreteElliptic.h"
16 #include "DataRecord.h"
17
18 //10 minutes at 1ms servo lp -> 300000
19 #define MAX_DATA_RECORDS_SIZE 300000
20
21 vector<string> DataThread::filterStrings,DataThread::bandWidthStrings,DataThread::
       orderStrings;
22 vector<string> DataThread::pbRipStrings,DataThread::sbRipStrings,DataThread::
       typeStrings;
23 vector<DataThread::CMInfo> DataThread::cmDataSet;
24
25 #define SEC_TO_MICRO 1000000UL
26
27 DataThread::DataThread()
28 {
29   stopped = false;
30   paused = false;
31   double dataLoopTime = (galilCon->getLoopTime())*(galilCon->getDataRate());
32
33   for(unsigned int i=0;i<cmDataSet.size();i++)
34   {
35     double bw = filterConfig[bandWidthStrings[i]];
36     int ord = (int)filterConfig[orderStrings[i]];
37     double pbr = filterConfig[pbRipStrings[i]];
38     double sbr = filterConfig[sbRipStrings[i]];
39     int type = (int) filterConfig[typeStrings[i]];
40
41     //cout << "Creating filter" << endl;
42     if(type == 0)
43     {
44       filters.push_back(new DiscreteButterworth(bw,ord,dataLoopTime));
45     }
46     else if (type==1)
47     {
48       filters.push_back(new DiscreteChebychev(bw,ord,pbr,dataLoopTime));
49     }
50     else
51     {
52       filters.push_back(new DiscreteElliptic(bw,ord,pbr,sbr,dataLoopTime));
53     }
54     //cout << "creating data " << endl;
55     inputData.push_back(new FFTData(dataLoopTime));
56     outputData.push_back(new FFTData(dataLoopTime));
57   }
58
59
60   struct timezone tz;
61     struct timeval tv;
62
63     gettimeofday(&tv, &tz);
64     secBase = tv.tv_sec;
65     microSecBase = tv.tv_usec;
66 }
67
68 void DataThread::pause()
69 {
70   if(!stopped)
```

```
71    {
72      paused=true;
73    }
74  }
75  void DataThread::resume()
76  {
77    if(!stopped)
78    {
79      paused=false;
80    }
81  }
82  void DataThread::initializeFilterData()
83  {
84    //cout << "Filter Data Start" << endl;
85
86    filterStrings.     push_back("Left Knee");
87    bandWidthStrings.  push_back("AN4_FIL_CUTOFF");
88    orderStrings.    push_back("AN4_FIL_ORDER");
89    pbRipStrings.    push_back("AN4_FIL_PBRIP");
90    sbRipStrings.    push_back("AN4_FIL_SBRIP");
91    typeStrings.       push_back("AN4_FIL_TYPE");
92    cmDataSet.       push_back(CMInfo(CM_AN4        ,ANALOG_4     ,ABSOLUTE ));
93
94    filterStrings.     push_back("Right Knee");
95    bandWidthStrings.  push_back("AN1_FIL_CUTOFF");
96    orderStrings.    push_back("AN1_FIL_ORDER");
97    pbRipStrings.    push_back("AN1_FIL_PBRIP");
98    sbRipStrings.    push_back("AN1_FIL_SBRIP");
99    typeStrings.       push_back("AN1_FIL_TYPE");
100   cmDataSet.       push_back(CMInfo(CM_AN1        ,ANALOG_1     ,ABSOLUTE ));
101
102   filterStrings.     push_back("Left Hip");
103   bandWidthStrings.  push_back("AN2_FIL_CUTOFF");
104   orderStrings.    push_back("AN2_FIL_ORDER");
105   pbRipStrings.    push_back("AN2_FIL_PBRIP");
106   sbRipStrings.    push_back("AN2_FIL_SBRIP");
107   typeStrings.       push_back("AN2_FIL_TYPE");
108   cmDataSet.       push_back(CMInfo(CM_AN2        ,ANALOG_2     ,ABSOLUTE ));
109
110   filterStrings.     push_back("Right Hip");
111   bandWidthStrings.  push_back("AN3_FIL_CUTOFF");
112   orderStrings.    push_back("AN3_FIL_ORDER");
113   pbRipStrings.    push_back("AN3_FIL_PBRIP");
114   sbRipStrings.    push_back("AN3_FIL_SBRIP");
115   typeStrings.       push_back("AN3_FIL_TYPE");
116   cmDataSet.       push_back(CMInfo(CM_AN3        ,ANALOG_3     ,ABSOLUTE ));
117
118
119   filterStrings.     push_back("Left Knee Velocity");
120   bandWidthStrings.  push_back("L_KNEE_FIL_CUTOFF");
121   orderStrings.    push_back("L_KNEE_FIL_ORDER");
122   pbRipStrings.    push_back("L_KNEE_FIL_PBRIP");
123   sbRipStrings.    push_back("L_KNEE_FIL_SBRIP");
124   typeStrings.       push_back("L_KNEE_FIL_TYPE");
125   cmDataSet.       push_back(CMInfo(CM_LKNEE_VELOCITY   ,ANALOG_4_FIL     ,DERIVATIVE ));
126
127   filterStrings.     push_back("Right Knee Velocity");
128   bandWidthStrings.  push_back("R_KNEE_FIL_CUTOFF");
129   orderStrings.    push_back("R_KNEE_FIL_ORDER");
130   pbRipStrings.    push_back("R_KNEE_FIL_PBRIP");
131   sbRipStrings.    push_back("R_KNEE_FIL_SBRIP");
132   typeStrings.       push_back("R_KNEE_FIL_TYPE");
133   cmDataSet.       push_back(CMInfo(CM_RKNEE_VELOCITY   ,ANALOG_1_FIL     ,DERIVATIVE ));
134
135   filterStrings.     push_back("Left Hip Velocity");
```

```
136   bandWidthStrings. push_back("L_HIP_FIL_CUTOFF");
137   orderStrings.    push_back("L_HIP_FIL_ORDER");
138   pbRipStrings.    push_back("L_HIP_FIL_PBRIP");
139   sbRipStrings.    push_back("L_HIP_FIL_SBRIP");
140   typeStrings.       push_back("L_HIP_FIL_TYPE");
141
142   cmDataSet.       push_back(CMInfo(CM_LHIP_VELOCITY     ,ANALOG_2_FIL     ,DERIVATIVE ));
143
144   filterStrings.     push_back("Right Hip Velocity");
145   bandWidthStrings. push_back("R_HIP_FIL_CUTOFF");
146   orderStrings.    push_back("R_HIP_FIL_ORDER");
147   pbRipStrings.    push_back("R_HIP_FIL_PBRIP");
148   sbRipStrings.    push_back("R_HIP_FIL_SBRIP");
149   typeStrings.       push_back("R_HIP_FIL_TYPE");
150   cmDataSet.       push_back(CMInfo(CM_RHIP_VELOCITY  ,ANALOG_3_FIL     ,DERIVATIVE ));
151
152   filterStrings.     push_back("Roll Velocity");
153   bandWidthStrings. push_back("ROLL_FIL_CUTOFF");
154   orderStrings.    push_back("ROLL_FIL_ORDER");
155   pbRipStrings.    push_back("ROLL_FIL_PBRIP");
156   sbRipStrings.    push_back("ROLL_FIL_SBRIP");
157   typeStrings.       push_back("ROLL_FIL_TYPE");
158   cmDataSet.       push_back(CMInfo(CM_ROLL_VELOCITY  ,ROLL_CLICKS    ,DERIVATIVE ));
159
160   filterStrings.     push_back("Yaw Velocity");
161   bandWidthStrings. push_back("YAW_FIL_CUTOFF");
162   orderStrings.    push_back("YAW_FIL_ORDER");
163   pbRipStrings.    push_back("YAW_FIL_PBRIP");
164   sbRipStrings.    push_back("YAW_FIL_SBRIP");
165   typeStrings.       push_back("YAW_FIL_TYPE");
166   cmDataSet.       push_back(CMInfo(CM_YAW_VELOCITY    ,YAW_CLICKS     ,DERIVATIVE ));
167
168
169   filterStrings.     push_back("Pitch Velocity");
170   bandWidthStrings. push_back("PITCH_FIL_CUTOFF");
171   orderStrings.    push_back("PITCH_FIL_ORDER");
172   pbRipStrings.    push_back("PITCH_FIL_PBRIP");
173   sbRipStrings.    push_back("PITCH_FIL_SBRIP");
174   typeStrings.       push_back("PITCH_FIL_TYPE");
175   cmDataSet.       push_back(CMInfo(CM_PITCH_VELOCITY    ,PITCH_CLICKS  ,DERIVATIVE ));
176   //cout << "Filter Data END" << endl;
177 }
178 void DataThread::stop()
179 {
180   stopped = true;
181   paused=false;
182   dataMutex.lock();
183   currentDataLock.lockForWrite();
184   currentData = NULL;
185   unsigned int x;
186   for(x=0;x<dataRecords.size();x++)
187   {
188     delete dataRecords[x];
189   }
190   dataRecords.clear();
191   currentDataLock.unlock();
192   dataMutex.unlock();
193 }
194 void DataThread::saveLog()
195 {
196   unsigned int x;
197   FILE * outPtr = fopen("log","w");
198   for(x=0; x<logTimes.size();x++)
199   {
200     fprintf(outPtr,"%5lf %s\n",logTimes[x],logMessages[x].c_str());
```

```
201     }
202     fclose(outPtr);
203 }
204 void DataThread::restart()
205 {
206     stop();
207     wait();
208     DataRecord::initializeDataInfo();
209 
210     double dataLoopTime = (galilCon->getLoopTime())*(galilCon->getDataRate());
211     //cout <<"Data Loop Time" << dataLoopTime << endl;
212 
213     for(unsigned int i=0;i<cmDataSet.size();i++)
214     {
215       delete filters[i];
216       //cout << filterStrings[i] << "=" << filterConfig[bandWidthStrings[i]] << endl;
217       double bw = filterConfig[bandWidthStrings[i]];
218       int ord = (int)filterConfig[orderStrings[i]];
219       double pbr = filterConfig[pbRipStrings[i]];
220       double sbr = filterConfig[sbRipStrings[i]];
221       int type = (int) filterConfig[typeStrings[i]];
222 
223       if(type == 0)
224       {
225         filters[i] = new DiscreteButterworth(bw,ord,dataLoopTime);
226       }
227       else if (type==1)
228       {
229         filters[i] = new DiscreteChebychev(bw,ord,pbr,dataLoopTime);
230       }
231       else
232       {
233         filters[i] = new DiscreteElliptic(bw,ord,pbr,sbr,dataLoopTime);
234       }
235 
236       inputData[i]->setPeriod(dataLoopTime);
237       outputData[i]->setPeriod(dataLoopTime);
238     }
239     struct timezone tz;
240       struct timeval tv;
241 
242       gettimeofday(&tv, &tz);
243       secBase = tv.tv_sec;
244       microSecBase = tv.tv_usec;
245       stopped = false;
246       paused =false;
247       logMessages.clear();
248       logTimes.clear();
249       cout << "Data Thread Restarted" << endl;
250     start();
251 }
252 
253 void DataThread::runFilterFFTs(int i)
254 {
255     inputData[i]->runFFT();
256     outputData[i]->runFFT();
257 }
258 QwtData& DataThread::getFilterInput(int i)
259 {
260     return (QwtData&)  (*inputData[i]);
261 }
262 QwtData& DataThread::getFilterOutput(int i)
263 {
264     return (QwtData&) (*outputData[i]);
265 }
```

```
266 void DataThread::addLog(const string &s)
267 {
268   double t = systemTime();
269   logMessages.push_back(s);
270   logTimes.push_back(t);
271 }
272 double DataThread::systemTime()
273 {
274   unsigned long microTime;
275   struct timezone tz;
276     struct timeval tv;
277   gettimeofday(&tv, &tz);
278   microTime = (tv.tv_sec-secBase)*SEC_TO_MICRO+tv.tv_usec-microSecBase;
279   return ((double)microTime)/1000000.;
280 }
281 void DataThread::run()
282 {
283   //cout << "Data Thread Now Running" << endl;
284   charVector r;
285   int dt;
286   vector<char> * charVecPtr = (vector<char> *) &r;
287   DataRecord * rec;
288   CMInfo cmInfo;
289   int prevTime=0,currentTime;
290   double velocity,filteredOutput;
291   double currentReading;
292
293   struct timezone tz;
294     struct timeval tv;
295     double Q[4];
296     double servoLoopTime=galilCon->getLoopTime();
297
298     double dataUpdTime=(galilCon->getDataRate())*servoLoopTime;
299     int dataMTime = dataUpdTime*1000;
300     int dataRate = (galilCon->getDataRate());
301
302   vector<double> prevData(cmDataSet.size(),0);
303   //cout << "Maybe we are stopped? " << stopped << endl;
304   while(!stopped)
305   {
306     try
307     {
308       //cout << "Waiting for data " << endl;
309 #ifndef SIMULATOR
310       (*charVecPtr) = galilCon->getRecord();
311       gettimeofday(&tv,&tz);
312 #else
313         timespec ts;
314         gettimeofday(&tv,&tz);
315         ts.tv_nsec=tv.tv_usec*1000 +dataMTime*1000000;
316         ts.tv_sec=tv.tv_sec+ts.tv_nsec/1000000000;
317         ts.tv_nsec%=1000000000;
318       pthread_mutex_t junkMutex = PTHREAD_MUTEX_INITIALIZER;
319       pthread_cond_t junkCond = PTHREAD_COND_INITIALIZER;
320       pthread_cond_timedwait (&junkCond,&junkMutex,&ts);
321       mainWindow->bipedSimulator->incTime(dataRate);
322 #endif
323       //cout << "Got Data" << endl;
324       rec = new DataRecord;
325       rec->setGalilCharData(r);
326       rec->setContinuousData(CM_SYSTEM_TIME, systemTime());
327       rec->setContinuousData(CM_SM_COMPLETE,0);
328
329       Q[0]=.386*pow(rec->getReading(LHIP_MOT_CURRENT),2.);
330       Q[1]=.386*pow(rec->getReading(LKNEE_MOT_CURRENT),2.);
```

```
331         Q[2]=.386*pow(rec->getReading(RHIP_MOT_CURRENT),2.);
332         Q[3]=.386*pow(rec->getReading(RKNEE_MOT_CURRENT),2.);
333
334         //cout << dataUpdTime << endl;
335         //cout << "Attempting to addData" << endl;
336         //ThreadCom::shmPtr->addData(Q,tv,dataUpdTime);
337         //cout << "Add Complete" << endl;
338
339         currentTime = (int) rec->getReading(GALIL_TIME);
340         dt=(currentTime-prevTime);
341         if(dt<0)
342         {
343           dt+=65536;
344
345         }
346         //cout << "Going into Loop" << endl;
347         unsigned int i;
348         for(i=0;i<cmDataSet.size();i++)
349         {
350           cmInfo = cmDataSet[i];
351
352           currentReading  = rec->getReading(cmInfo.dReadingNum);
353           if(cmInfo.type == DERIVATIVE)
354           {
355             velocity     = (currentReading-prevData[i])/(dt*servoLoopTime);
356             inputData[i]->addData(velocity);
357             filteredOutput = filters[i]->input(velocity);
358           }
359           else
360           {
361             inputData[i]->addData(currentReading);
362             filteredOutput = filters[i]->input(currentReading);
363           }
364           outputData[i]->addData(filteredOutput);
365           rec->setContinuousData(cmInfo.cReadingNum,filteredOutput);
366           prevData[i]=currentReading;
367         }
368
369
370
371         //rec->setContinuousData(CM_JUMPING_STATE,mainStateMachine->getState());
372         if(!paused)
373         {
374           //printf("D) Waiting for data mutex\n");
375           dataMutex.lock();
376           //printf("D) Data Mutex Acquired\n");c
377           dataRecords.push_back(rec);
378           while(dataRecords.size()>MAX_DATA_RECORDS_SIZE)
379           {
380             delete dataRecords.front();
381             dataRecords.pop_front();
382           }
383           //printf("D) Data Mutex Unlocking\n");
384           currentData = rec;
385           dataMutex.unlock();
386           newDataCondition.wakeAll();
387         }
388
389         prevTime=currentTime;
390       }
391       catch(std::string errStr)
392       {
393         std::cout << errStr;
394       }
395     }
```

```
396     stopped = false;
397 }
```

ReadingDefines.h

```
 1 #ifndef READINGDEFINES_H_
 2 #define READINGDEFINES_H_
 3 #include "GlobalDefines.h"
 4
 5 enum Reading {
 6     JUMPING_STATE,
 7     SM_COMPLETE,
 8     LHIP_POSITION_ANGLE,
 9     LHIP_VELOCITY_ANGLE,
10     LHIP_MOT_POSITION_ANGLE,
11     LHIP_MOT_CURRENT,
12     LHIP_MOT_TORQUE,
13     LHIP_SEA_DEFLECTION,
14     LHIP_SEA_TORQUE,
15     LHIP_SEA_ENGERGY,
16
17     LKNEE_POSITION_ANGLE,
18     LKNEE_VELOCITY_ANGLE,
19     LKNEE_MOT_POSITION_ANGLE,
20     LKNEE_MOT_CURRENT,
21     LKNEE_MOT_TORQUE,
22     LKNEE_SEA_DEFLECTION,
23     LKNEE_SEA_TORQUE,
24     LKNEE_SEA_ENGERGY,
25
26     LLEG_LENGTH,
27     LLEG_ANGLE,
28
29     RHIP_POSITION_ANGLE,
30     RHIP_VELOCITY_ANGLE,
31     RHIP_MOT_POSITION_ANGLE,
32     RHIP_MOT_CURRENT,
33     RHIP_MOT_TORQUE,
34     RHIP_SEA_DEFLECTION,
35     RHIP_SEA_TORQUE,
36     RHIP_SEA_ENGERGY,
37
38     RKNEE_POSITION_ANGLE,
39     RKNEE_VELOCITY_ANGLE,
40     RKNEE_MOT_POSITION_ANGLE,
41     RKNEE_MOT_CURRENT,
42     RKNEE_MOT_TORQUE,
43     RKNEE_SEA_DEFLECTION,
44     RKNEE_SEA_TORQUE,
45     RKNEE_SEA_ENGERGY,
46
47     RLEG_LENGTH,
48     RLEG_ANGLE,
49
50
51     ROLL_ANGLE,
52     ROLL_VELOCITY_ANGLE,
53     YAW_ANGLE,
54     YAW_VELOCITY_ANGLE,
55     PITCH_ANGLE,
56     PITCH_VELOCITY_ANGLE,
57
```

```
58
59      GALIL_TIME,
60      SYSTEM_TIME,
61      ANALOG_1,
62      ANALOG_2,
63      ANALOG_3,
64      ANALOG_4,
65
66      ANALOG_1_FIL,
67      ANALOG_2_FIL,
68      ANALOG_3_FIL,
69      ANALOG_4_FIL,
70
71      ANALOG_1_NUM,
72      ANALOG_2_NUM,
73      ANALOG_3_NUM,
74      ANALOG_4_NUM,
75
76      IN_1 ,
77      IN_2 ,
78
79      LFOOT_CONT,
80      LFOOT_HEIGHT,
81      RFOOT_CONT,
82      RFOOT_HEIGHT,
83
84
85      LHIP_OFF,
86      LHIP_ON,
87      LHIP_MOV,
88      LHIP_VELOCITY_VOLTS,
89      LHIP_MOT_POSITION_CLICKS,
90      LHIP_AMP_COMMAND,
91
92      LKNEE_OFF,
93      LKNEE_ON,
94      LKNEE_MOV,
95      LKNEE_VELOCITY_VOLTS,
96      LKNEE_MOT_POSITION_CLICKS,
97      LKNEE_AMP_COMMAND,
98
99      RHIP_OFF,
100     RHIP_ON,
101     RHIP_MOV,
102     RHIP_VELOCITY_VOLTS,
103     RHIP_MOT_POSITION_CLICKS,
104     RHIP_AMP_COMMAND,
105
106     RKNEE_OFF,
107     RKNEE_ON,
108     RKNEE_MOV,
109     RKNEE_VELOCITY_VOLTS,
110     RKNEE_MOT_POSITION_CLICKS,
111     RKNEE_AMP_COMMAND,
112
113     ROLL_CLICKS,
114     ROLL_VELOCITY_CLICKS,
115     YAW_CLICKS,
116     YAW_VELOCITY_CLICKS,
117     PITCH_CLICKS,
118     PITCH_VELOCITY_CLICKS,
119
120     HIP_HEIGHT,
121
122     LHIP_MOT_DESIRED_ANGLE,
```

```
123     LKNEE_MOT_DESIRED_ANGLE,
124     RHIP_MOT_DESIRED_ANGLE,
125     RKNEE_MOT_DESIRED_ANGLE,
126
127     LHIP_MOT_DESIRED_CLICKS,
128     LKNEE_MOT_DESIRED_CLICKS,
129     RHIP_MOT_DESIRED_CLICKS,
130     RKNEE_MOT_DESIRED_CLICKS,
131     RHIP_AMP_STATUS,
132     RHIP_AMP_CURRENT,
133     NUM_READINGS
134 };
135
136 enum ContMeasmnt {
137     CM_AN1,
138     CM_AN2,
139     CM_AN3,
140     CM_AN4,
141
142     CM_LHIP_VELOCITY,
143     CM_LKNEE_VELOCITY,
144     CM_RHIP_VELOCITY,
145     CM_RKNEE_VELOCITY,
146
147     CM_ROLL_VELOCITY,
148     CM_YAW_VELOCITY,
149     CM_PITCH_VELOCITY,
150
151
152     CM_SYSTEM_TIME,
153     CM_JUMPING_STATE,
154     CM_SM_COMPLETE,
155     CM_SIZE
156 };
157
158 enum GalilReading {
159     GR_TPA,
160     GR_TPB,
161     GR_TPC,
162     GR_TPD,
163     GR_TDA,
164     GR_TDB,
165     GR_TDC,
166     GR_TIME,
167     GR_AN1,
168     GR_AN2,
169     GR_AN3,
170     GR_AN4,
171     GR_IN1,
172     GR_IN2,
173     GR_MOA,
174     GR_MOB,
175     GR_MOC,
176     GR_MOD,
177     GR_BGA,
178     GR_BGB,
179     GR_BGC,
180     GR_BGD,
181     GR_TTA,
182     GR_TTB,
183     GR_TTC,
184     GR_TTD,
185     GR_RPA,
186     GR_RPB,
187     GR_RPC,
```

```
188     GR_RPD,
189     GR_IN3,
190     GR_ZAA,
191     GR_SIZE
192 };
193
194 #endif /*READINGDEFINES_H_*/
```

DataRecord.h

```
 1 #ifndef DATARECORD_H_
 2 #define DATARECORD_H_
 3 #include <vector>
 4 #include <string>
 5 #include <map>
 6 #include "ReadingDefines.h"
 7 #include "GlobalClasses.h"
 8
 9 using namespace std;
10
11 class DataRecord
12 {
13   enum DataType {
14     SPECIAL,
15     DIRECT,
16     CALCULATED,
17     CONVERTED,
18   };
19
20
21   typedef double (DataRecord::*PtrToMemberType)();
22
23   typedef struct DataInfoStruct
24   {
25     DataInfoStruct(string n,string u,DataType t,PtrToMemberType p) : name(n),units(u)
         ,type(t),calcFnPtr(p),readingTypeNumber(0),conversionFactor(0),offset(0) {};
26     DataInfoStruct(string n,string u,DataType t,int r,double c) : name(n),units(u),
         type(t),calcFnPtr(NULL),readingTypeNumber(r),conversionFactor(c),offset(0)
         {};
27     DataInfoStruct(string n,string u,DataType t,int r,double c,double o) : name(n),
         units(u),type(t),calcFnPtr(NULL),readingTypeNumber(r),conversionFactor(c),
         offset(o) {};
28     DataInfoStruct(string n,string u,DataType t,int r) : name(n),units(u),type(t),
         calcFnPtr(NULL),readingTypeNumber(r),conversionFactor(0),offset(0) {};
29     DataInfoStruct(string n,string u,DataType t) : name(n),units(u),type(t),calcFnPtr
         (NULL),readingTypeNumber(0),conversionFactor(0),offset(0) {};
30     DataInfoStruct() : name(""),units(""),type(SPECIAL),calcFnPtr(NULL),
         readingTypeNumber(0),conversionFactor(0),offset(0) {};
31     string name;
32     string units;
33     DataType type;
34     PtrToMemberType calcFnPtr;
35     int readingTypeNumber;
36     double conversionFactor;
37     double offset;
38   } DataInfo;
39
40
41
42
43   public:
44     DataRecord();
```

```
45      void    setGalilCharData(charVector&);
46      double getReading(int);
47      double getGalilReading(string);
48      void setContinuousData(ContMeasmnt,double);
49      double getAxisVelocity(int);
50      double getAxisPosition(int);
51
52      static void setOffset(Reading r,double o);
53      static void initializeDataInfo();
54      static vector<string> reqGalilData;
55      static vector<int> galilMins;
56      static vector<int> galilMaxs;
57      static vector<string> readingStrings;
58      static vector<string> unitStrings;
59
60    private:
61      static int velocityReadings[5];
62      static int positionReadings[5];
63      charVector galilCharData;
64      doubleVector galilRawData;
65      doubleVector contData;
66
67
68      void populateGalilMeasurements();
69      void populateUserVariables();
70
71
72      double cLHipEnergy();
73      double cLKneeEnergy();
74      double cRHipEnergy();
75      double cRKneeEnergy();
76
77      double cLHipDeflection();
78      double cLKneeDeflection();
79      double cRHipDeflection();
80      double cRKneeDeflection();
81
82      double cLLegLength();
83      double cLLegAngle();
84      double cRLegLength();
85      double cRLegAngle();
86
87      double cLFootHeight();
88      double cRFootHeight();
89      double cHipHeight();
90
91      static map<int,DataInfo> dataTypeMap;
92  };
93  #endif /*DATARECORD_H_*/
```

F.2 Digital Filtering Files

DiscreteFilter.h

```
1  #ifndef DISCRETEFILTER_H_
2  #define DISCRETEFILTER_H_
3
4  #include "GlobalClasses.h"
5
6  class DiscreteFilter
7  {
```

```
 8   public:
 9     DiscreteFilter(double f, int N, double t);
10     double input(double x);
11     double getMaxPole();
12     double getOutput();
13     void clear();
14     double magFreqResponse(double f);
15   protected:
16     doubleVector num;
17     doubleVector den;
18     double T;
19     double maxPole;
20     double Wca,WcDig;
21     int n;
22
23     void conv(complexDoubleVector &a,complexDoubleVector &b);
24
25   private:
26     doubleVector yp;
27     doubleVector xp;
28     double output;
29     int pos;
30 };
31 #endif /*DISCRETEFILTER_H_*/
```

DiscreteFilter.cpp

```
 1 #include "DiscreteFilter.h"
 2 #define PI 3.14159265358979323846
 3 #include <iostream>
 4 using namespace std;
 5
 6 DiscreteFilter::DiscreteFilter(double freqCutoffHertz, int N, double t)
 7 {
 8   n=N;
 9   T=t;
10   output = 0;
11   pos = 0;
12
13   //Set our previous inputs and outputs to 0
14   //(Used with transfer function to find output)
15   yp.resize(n+1,0);
16   xp.resize(n+1,0);
17
18   //Due to frequency warping of bi-linear transform (see Wikipedia)
19   //We calculate our analog cutoff frequency as below
20   WcDig = 2.*PI*freqCutoffHertz;
21
22   //If our cutoff frequency is more than half the sampling frequency we have issues
23   //i.e. our cutoff frequency will get "aliased" to a different cuttoff frequency
24   if(freqCutoffHertz > 1/(2*T))
25   {
26       //At the request of Nyquist
27       cout << "Cutoff Frequency " << freqCutoffHertz << " too high for sampling rate"
              << endl;
28   }
29
30   Wca = 2/T * tan(WcDig*(T/2));
31   //cout << "WCA" << Wca << endl;
32 }
33 void DiscreteFilter::clear()
34 {
```

```
35    output = 0;
36    pos = 0;
37    for(int i=0;i<=n;i++)
38    {
39      xp[i]=0;
40      yp[i]=0;
41    }
42  }
43
44  double DiscreteFilter::input(double x)
45  {
46    int k;
47    int y=pos;
48    output=0;
49    xp[pos]=x;
50    yp[pos]=0;
51    for(k=0;k<=n;k++)
52    {
53      //cout << "x[n-" << k << "]\t" <<  xp[y] << endl;
54      //cout << "y[n-" << k << "]\t" <<  yp[y] << endl;
55      output+=num[k]*xp[y]-den[k]*yp[y];
56      y++;
57      y%=(n+1);
58    }
59    yp[pos]=output;
60
61    if(pos==0)
62      pos=n;
63    else
64      pos--;
65
66    return output;
67  }
68  double DiscreteFilter::getMaxPole()
69  {
70      return maxPole;
71  }
72  double DiscreteFilter::magFreqResponse(double f)
73  {
74    complex <double> z = polar(1.,2*PI*f*T),denSum(0,0),numSum(0,0);
75
76     for(unsigned int d=0;d<den.size();d++)
77     {
78      double power = (double) (den.size()-1 -d);
79      denSum+=pow(z,power)*den[d];
80     }
81     for(unsigned int i=0;i<num.size();i++)
82     {
83      double power = (double) (num.size()-1 -i);
84      numSum+=pow(z,power)*num[i];
85     }
86     return abs(numSum/denSum);
87  }
88
89  double DiscreteFilter::getOutput()
90  {
91    return output;
92  }
93
94  void DiscreteFilter::conv(complexDoubleVector &a,complexDoubleVector &b)
95  {
96    unsigned int x,y;
97    unsigned int totalSize = a.size() + b.size() - 1;
98    complexDoubleVector ans;
99    for(x=0;x<totalSize;x++)
```

```
100     {
101       ans.push_back(0);
102     }
103
104     for(x=0;x<a.size();x++)
105     {
106       for(y=0;y<b.size();y++)
107         ans[x+y]+=a[x]*b[y];
108     }
109     ans.swap(a);
110 }
```

DiscreteButterworth.h

```
1 #ifndef DISCRETEBUTTERWORTH_H_
2 #define DISCRETEBUTTERWORTH_H_
3 #include "GlobalClasses.h"
4 #include "DiscreteFilter.h"
5 using namespace std;
6
7 class DiscreteButterworth : public DiscreteFilter
8 {
9   public:
10     DiscreteButterworth(double Fc, int N, double t);
11 };
12 #endif /*DISCRETEBUTTERWORTH_H_*/
```

DiscreteButterworth.cpp

```
1 #include "DiscreteButterworth.h"
2 #include <complex>
3 #include <math.h>
4 #include <cstdio>
5 #include <iostream>
6 using namespace std;
7
8 #define PI 3.14159265358979323846
9
10 DiscreteButterworth::DiscreteButterworth(double freqCutoffHertz, int N, double t) :
       DiscreteFilter(freqCutoffHertz,N,t)
11 {
12   int x;
13   //cout << freqCutoffHertz << "-" << N << "-" << t << endl;
14   //Vectors for analog an digital poles
15   //We will first find analog poles from the analog cutoff frequency derived above
16   //Then via the bi-linear transformation well find the digital poles
17   //Note: For a digitial butter-worth all the ZEROs lie at -1
18   vector<complex <double> > PolesA;
19   vector<complex <double> > PolesD;
20
21   //Poles for butterworth are evenly spaced in LHP at a radius of the cutoff requency
22   // i.e. angle between each pole is PI/n, and angle from the imaginary axis to the
         first
23   //       and last poles is PI/(2*n)
24   double angle;
25   angle = PI/2 + PI/(2*n);
26
27   //Keep track of dominant pole
```

```
28    maxPole=0;
29
30    //Loop through placing each pole
31    for(x=0 ; x< n ; x++)
32    {
33      PolesA.push_back(polar(Wca,angle));
34
35      //Use BiLinear Transform to find digital poles
36      PolesD.push_back( (2/T + PolesA[x]) / (2/T - PolesA[x]) );
37
38      //cout << "Pole D " << x << " " << PolesD.back() << endl;
39
40      if(abs(PolesD.back()) > maxPole)
41      {
42        maxPole = abs(PolesD.back());
43      }
44
45      angle+=PI/n;
46    }
47
48    //Construct Pascals triangle to find polynomial coefficients for numerator (z+1)^n
49    doubleVector oldCoefficients;
50    num.clear();
51
52    //We start with (z+1)^1
53    num.push_back(1);
54    num.push_back(1);
55
56    //Then build up layer by layer
57    //OldCoefficients holds the previous layer and num holds the current layer
58    for(int x=2;x<=n;x++)
59    {
60      num.swap(oldCoefficients);
61      int y;
62      num.clear();
63      //Pascals triangale always starts with 1 on each level
64      num.push_back(1);
65
66      //And then uses the neighboring entries on the higher level to fill in up till
            the last entry
67      for(y=1;y<x;y++)
68      {
69        num.push_back(oldCoefficients[y-1]+oldCoefficients[y]);
70      }
71
72      //Then the last entry is always 1
73      num.push_back(1);
74    }
75
76    //printf("Pascal Complete\n");
77
78    //singleTerm holds the coefficients for a single digitial pole
79    complexDoubleVector singleTerm;
80
81    //cDen holds the coefficients for the mutiplied out denominator
82    complexDoubleVector cDen;
83
84    //the denominator starts out as 1 [we will mutiply it by each pole term, (z+pole_k)
          ]
85    cDen.push_back(1);
86
87    //initialize the single pole term as z+1
88    singleTerm.push_back(1);
89    singleTerm.push_back(1);
90    for(x=0;x<n;x++)
```

```
91    {
92      //Change the pole term to (z+pole_k)
93      singleTerm[1]=-1.*PolesD[x];
94
95      //Mutlitply the denominator by that pole term
96      conv(cDen,singleTerm);
97    }
98
99    //Now construct find gain coefficient needed for unity DC gain
100   double gain;
101   double sumNum=0;
102   double sumDen=0;
103   den.clear();
104   for(x=0;x<=n;x++)
105   {
106     //Since there is some rounding error, only extract the real parts from the
          mutiplied
107     //out denominator found previously
108     den.push_back(real(cDen[x]));
109     sumNum+=num[x];
110     sumDen+=den[x];
111     //cout << "Den " << x << " => " << den[x] << endl;
112   }
113
114   //Find gain from sums (note: this is due to DC digital frequency z=1)
115   gain = sumDen/sumNum;
116
117   //Scale the numerator
118   for(x=0;x<=n;x++)
119   {
120     num[x]*=gain;
121     //cout << "Num " << x << " => " << num[x] << endl;
122   }
123   //printf("Done\n");
124 }
```

F.3 State Machine Files

StateMachine.h

```
1 #ifndef STATEMACHINE_H_
2 #define STATEMACHINE_H_
3 #include <QWidget>
4 #include <vector>
5 #include <string>
6 class FuzzyController;
7 class DataRecord;
8 class QPainter;
9 using namespace std;
10 class StateMachine : public QWidget
11 {
12   Q_OBJECT
13
14 public:
15   StateMachine(QWidget * parent =0);
16   void draw(QPainter *painter);
17   virtual void checkState(string,DataRecord *) = 0;
18   bool isRunning();
19   QSize sizeHint() const;
20   int getState();
21   void setState(int);
```

```
22 public slots:
23   void start();
24   void stop();
25 protected:
26   string intToString(int);
27   string doubleToString(double);
28   void paintEvent(QPaintEvent * event);
29   double abs(double);
30   bool found(string,string);
31   vector<int> states;
32   vector<QString> stateStrings;
33
34   FuzzyController * fuzzyController;
35
36   volatile int state;
37   int initState;
38   volatile bool running;
39 };
40 #endif /*STATEMACHINE_H_*/
```

StateMachine.cpp

```
 1 #include "StateMachine.h"
 2 #include <iostream>
 3 #include <string>
 4 #include <sstream>
 5 #include <QPainter>
 6 #include <QPaintEvent>
 7
 8 #define ITEM_WIDTH  60
 9 #define ITEM_HEIGHT 40
10 #define ITEM_PADDING 10
11
12 StateMachine::StateMachine(QWidget * parent) : QWidget(parent)
13 {
14   running = false;
15 }
16 bool StateMachine::isRunning()
17 {
18   return running;
19 }
20 void StateMachine::stop()
21 {
22   if(running)
23   {
24     state=initState;
25     running=false;
26   }
27 }
28 void StateMachine::setState(int s)
29 {
30   state = s;
31 }
32 int StateMachine::getState()
33 {
34   return state;
35 }
36 void StateMachine::start()
37 {
38   if(!running)
39   {
40     state=initState;
```

```
41      running=true;
42    }
43  }
44  QSize StateMachine::sizeHint() const
45  {
46    int totalWidth  = ITEM_PADDING+(ITEM_WIDTH+ITEM_PADDING)*states.size();
47    int totalHeight = 2*ITEM_PADDING+ITEM_HEIGHT;
48    QSize s(totalWidth,totalHeight);
49
50    return s*2;
51  }
52  void StateMachine::paintEvent(QPaintEvent * /*event*/)
53  {
54    QPainter painter(this);
55    int totalWidth  = ITEM_PADDING+(ITEM_WIDTH+ITEM_PADDING)*states.size()+ITEM_HEIGHT
          /2+ITEM_PADDING;
56    int totalHeight = 2*ITEM_PADDING+ITEM_HEIGHT;
57    painter.setWindow(0,0,totalWidth,totalHeight);
58
59    int sideWidth = qMin(width(), totalWidth*height()/totalHeight);
60    int sideHeight = sideWidth*totalHeight/totalWidth;
61    painter.setViewport((width() - sideWidth)/2,(height()-sideHeight)/2,sideWidth,
          sideHeight);
62    draw(&painter);
63  }
64
65  void StateMachine::draw(QPainter * painter)
66  {
67    QPen thinPen(Qt::black,2);
68    QFont bigFont;
69
70    int x=ITEM_PADDING;
71    if(running)
72    {
73      painter->setBrush(QBrush(QColor(0,255,0)));
74    }
75    else
76    {
77      painter->setBrush(QBrush(QColor(0,100,0)));
78    }
79    painter->drawEllipse(x,ITEM_PADDING+ITEM_HEIGHT/4,ITEM_HEIGHT/2,ITEM_HEIGHT/2);
80    x+=ITEM_PADDING+ITEM_HEIGHT/2;
81
82    bigFont.setPointSize(10);
83    painter->setPen(thinPen);
84    painter->setRenderHint(QPainter::Antialiasing,true);
85    painter->setPen(QPen(Qt::black,2,Qt::SolidLine,Qt::RoundCap));
86    painter->setFont(bigFont);
87    painter->setBrush(QBrush(Qt::white));
88
89    for(unsigned int i=0;i<states.size();i++)
90    {
91      if(((int)i)==state)
92      {
93        painter->setBrush(QBrush(Qt::gray));
94      }
95      else
96      {
97        painter->setBrush(QBrush(Qt::white));
98      }
99      painter->drawEllipse(x,ITEM_PADDING,ITEM_WIDTH,ITEM_HEIGHT);
100     //cout << "Painting state " << stateStrings[i].toStdString() << endl;
101     painter->drawText(x,ITEM_PADDING,ITEM_WIDTH,ITEM_HEIGHT,Qt::AlignHCenter | Qt::
            AlignCenter,stateStrings[i]);
102     x+=ITEM_WIDTH+ITEM_PADDING;
```

```
103   }
104 }
105 bool StateMachine::found(string a,string b)
106 {
107   return a.find(b)!=string::npos;
108 }
109 double StateMachine::abs(double a)
110 {
111   if(a<0)
112     return -a;
113   else
114     return a;
115 }
116 string StateMachine::intToString(int x)
117 {
118   string s;
119   stringstream ss;
120   ss << x;
121   ss >> s;
122   return s;
123 }
124 string StateMachine::doubleToString(double x)
125 {
126   string s;
127   stringstream ss;
128   ss << x;
129   ss >> s;
130   return s;
131 }
```

WalkingStateMachine.h

```
1 #ifndef WALKINGSTATEMACHINE_H_
2 #define WALKINGSTATEMACHINE_H_
3
4 #include "StateMachine.h"
5 #include "FuzzyDefines.h"
6 #include <QWidget>
7 class DataRecord;
8 class DiscreteButterworth;
9 class WalkingStateMachine : virtual public StateMachine
10 {
11   Q_OBJECT
12 public:
13   void checkState(string,DataRecord*);
14   WalkingStateMachine(QWidget * parent=0);
15 private:
16   //OutputMap outputs;
17   void InvKin(double cartPositions[], double linkPosition[],double motorPosition[]);
18   void FFLinkControl(double cartPositions[],double motorPositions[],double
         rightPercent);
19   void FFLegControl(double cartPositions[],double motorPositions[]);
20   void CubicSpline(double xo,double xo_dot, double xf,double xf_dot,double T,double
         params[]);
21
22
23   DiscreteButterworth * filters[4];
24   DiscreteButterworth * vFilters[4];
25   DataRecord * record;
26   double FFootPos[2][2];
27   double linkVel[4];
28   double linkAngle[4];
```

```
29    bool correct[4];
30    double correction[4];
31 };
32
33 #endif /*WALKINGSTATEMACHINE_H_*/
```

WalkingStateMachine.cpp

```
 1 #include "WalkingStateMachine.h"
 2 #include "GlobalVariables.h"
 3 #include "GuiUpdateThread.h"
 4 #include "GalilConnection.h"
 5 #include "DiscreteButterworth.h"
 6 #include "DataRecord.h"
 7 #include <iostream>
 8 #include <sstream>
 9 #include <string>
10 #include <QDateTime>
11 #include <QSlider>
12 #include <vector>
13
14 using namespace std;
15 #define g 9.81
16 #define pi PI
17 #define LINK_LENGTH THIGH_LENGTH
18 #define WEIGHT 12.1
19 #define Kd 0
20
21 enum WalkingStates
22 {
23    INIT,
24    POS,
25    RAISE,
26    HOLD,
27    STANCE_1,
28    SWING_L,
29    STANCE_2,
30    SWING_R
31 };
32
33
34 WalkingStateMachine::WalkingStateMachine(QWidget * parent) : StateMachine(parent)
35 {
36    initState = INIT;
37    state = INIT;
38
39    states.push_back(INIT);
40    stateStrings.push_back("Init");
41
42    states.push_back(POS);
43    stateStrings.push_back("Pos");
44
45    states.push_back(RAISE);
46    stateStrings.push_back("Raise");
47
48    states.push_back(HOLD);
49    stateStrings.push_back("Hold");
50
51    states.push_back(STANCE_1);
52    stateStrings.push_back("Stance 1");
53
54    states.push_back(SWING_L);
```

```
55    stateStrings.push_back("Swing L");
56
57    states.push_back(STANCE_2);
58    stateStrings.push_back("Stance 2");
59
60    states.push_back(SWING_R);
61    stateStrings.push_back("Swing R");
62
63    for(int i=0;i<4;i++)
64    {
65      filters[i]=new DiscreteButterworth(100,3,.001);
66    }
67
68    cout << "Walking State Machine Created!" << endl;
69  }
70  void WalkingStateMachine::checkState(string mesg,DataRecord * rec)
71  {
72    //cout << "State Machine" << endl;
73    record = rec;
74    static bool soundPlayed=false;
75    static double prevMot[4];
76    static double prevTime;
77    static double transitionTime;
78    static double startTime;
79    double cycles = 10;
80    static double cycleNum = 0;
81
82    double sysTime = rec->getReading(SYSTEM_TIME);
83    double timeFactor = (sysTime-startTime)/(transitionTime-startTime);
84
85    if(timeFactor>1)
86      timeFactor=1;
87
88
89    double currentTime   =  (int) rec->getReading(GALIL_TIME);
90    double servoLoopTime=galilCon->getLoopTime();
91    double dt=(currentTime-prevTime);
92    if(dt<0)
93    {
94      dt+=65536;
95    }
96    dt*=servoLoopTime;
97
98    double t1_l=rec->getReading(LHIP_POSITION_ANGLE);
99    double t2_l=t1_l+rec->getReading(LKNEE_POSITION_ANGLE);
100
101   double t1_r=rec->getReading(RHIP_POSITION_ANGLE);
102   double t2_r=t1_r+rec->getReading(RKNEE_POSITION_ANGLE);
103
104   FFootPos[0][0]=LINK_LENGTH*(sin(t1_l)+sin(t2_l));
105   FFootPos[0][1]=-LINK_LENGTH*(cos(t1_l)+cos(t2_l));
106
107   //guiUpdateThread->sendLog("L " + doubleToString(FFootPos[0][0]));
108
109   FFootPos[1][0]=LINK_LENGTH*(sin(t1_r)+sin(t2_r));
110   FFootPos[1][1]=-LINK_LENGTH*(cos(t1_r)+cos(t2_r));
111
112   //guiUpdateThread->sendLog("R " + doubleToString(FFootPos[1][0]));
113
114   linkAngle[0]=rec->getReading(LHIP_POSITION_ANGLE);
115   linkAngle[1]=rec->getReading(LKNEE_POSITION_ANGLE);
116   linkAngle[2]=rec->getReading(RHIP_POSITION_ANGLE);
117   linkAngle[3]=rec->getReading(RKNEE_POSITION_ANGLE);
118
119   linkVel[0]=rec->getReading(LHIP_VELOCITY_ANGLE);
```

```
120   linkVel[1]=linkVel[0]+rec->getReading(LKNEE_VELOCITY_ANGLE);
121   linkVel[2]=rec->getReading(RHIP_VELOCITY_ANGLE);
122   linkVel[3]=linkVel[2]+rec->getReading(RKNEE_VELOCITY_ANGLE);
123
124   double jointPositions[4],motorPositions[4],motorVelocities[4];
125   bool sendCommand=false;
126   bool interpVelocities=true;
127
128
129   static double speed_in_prev=.12;
130   static double speed_out_prev=.12;
131
132
133   double tau = mainWindow->tauSlider->value()/1000.;
134   double T = 2*servoLoopTime;
135
136   double speed_in = mainWindow->speedSlider->value()/100.;
137
138
139   //Correct for frequency warping
140   //tau = T/(2*atan(T/(2*tau)));
141
142   //double speed = (speed_in_prev + speed_in+(2*tau/T-1)*speed_out_prev)/(2*tau/T+1);
143   double speed = speed_in;
144   speed_out_prev = speed;
145   speed_in_prev=speed_in;
146
147   //guiUpdateThread->sendLog("Tau "+doubleToString(tau));
148   //guiUpdateThread->sendLog("Sp "+doubleToString(speed));
149
150
151   double stride_min=.12; //.15
152   double stride_max=.37; //.7
153   double duty_min=.53;
154   double duty_max=.8;
155   double tr_max=1.;
156   double speed_cutoff_low = (1-duty_max)*stride_min/tr_max;
157   double speed_cutoff_high = .5/2;
158   double dutyFactor;
159   double period;
160   static double phase=0;
161
162   if(abs(speed)<speed_cutoff_low)
163   {
164     dutyFactor=1-abs(speed)*tr_max/stride_min;
165     period=tr_max/(1-dutyFactor);
166   }
167   else if(abs(speed)<speed_cutoff_high)
168   {
169     double factor = (abs(speed)-speed_cutoff_low)/(speed_cutoff_high-speed_cutoff_low
          );
170     dutyFactor=duty_max*(1-factor)+duty_min*factor;
171     period = (stride_min*(1-factor)+stride_max*factor)/abs(speed);
172   }
173   else
174   {
175     dutyFactor=duty_min;
176     period=stride_max/abs(speed);
177   }
178
179   double stride = abs(speed)*period;
180   double stroke = dutyFactor*stride;
181
182   double forward = -stroke/2;
183   double backward = stroke/2;
```

```
184
185   double half_forward = -(stride-stroke)/2;
186   double half_backward = (stride-stroke)/2;
187
188   double halfStroke = stroke/2;
189
190   double h=sqrt(.495*.495-forward*forward);
191   double cartY=-h;
192   double stance_length = sqrt(cartY*cartY+half_forward*half_forward);
193
194   double liftHeight=mainWindow->liftSlider->value()/1000.;
195
196   double gamma=1.05;
197
198
199   double swing_time_1=2*(gamma-1)/(4*gamma-2);
200   double swing_time_2=1-swing_time_1;
201
202
203   if(state==INIT)
204   {
205     speed_in_prev = mainWindow->speedSlider->value()/100.;
206     speed=speed_in_prev;
207     speed_out_prev=speed;
208     cycleNum=0;
209     phase=0;
210     if(!calibrationStateMachine->isCalibComplete())
211     {
212       guiUpdateThread->sendLog("Calibration Not Complete");
213       GalilConnection::beep(false);
214       running=false;
215       return;
216     }
217     galilCon->command("XQ #INITPD,4");
218     galilCon->command("PT1,1,1,1;CMABCD;DT1;");
219     galilCon->command("AC25200000,25200000,25200000,25200000");
220     galilCon->command("DC25200000,25200000,25200000,25200000");
221
222     double cartPositions[4]={half_backward,cartY,forward,cartY};
223     //double cartPositions[4]={0,cartY,0,cartY};
224     cout << forward << "," << backward << endl;
225     InvKin(cartPositions,jointPositions,motorPositions);
226
227     cout << .25*sin(motorPositions[0])+.25*sin(motorPositions[1]) << endl;
228
229     motorVelocities[0]=.2;
230     motorVelocities[1]=.2;
231     motorVelocities[2]=.2;
232     motorVelocities[3]=.2;
233
234     sendCommand=true;
235     interpVelocities=false;
236     transitionTime=rec->getReading(SYSTEM_TIME)+4.;
237     state=POS;
238   }
239   else if (state==POS)
240   {
241     if(rec->getReading(SYSTEM_TIME)>transitionTime)
242     {
243       if(!soundPlayed)
244       {
245         GalilConnection::notify();
246         soundPlayed=true;
247       }
248
```

```
249         transitionTime=rec->getReading(SYSTEM_TIME)+2.;
250         state=HOLD;
251       }
252     }
253     else if (state==HOLD)
254     {
255
256       if(rec->getReading(SYSTEM_TIME)>transitionTime)
257       {
258         state=STANCE_1;
259         prevMot[0]=rec->getReading(LHIP_MOT_POSITION_ANGLE);
260         prevMot[1]=rec->getReading(LKNEE_MOT_POSITION_ANGLE);
261         prevMot[2]=rec->getReading(RHIP_MOT_POSITION_ANGLE);
262         prevMot[3]=rec->getReading(RKNEE_MOT_POSITION_ANGLE);
263
264       }
265       //running=false;
266     }
267     else
268     {
269       double segLeft = 511-rec->getGalilReading("_CM");
270       /*static double iErr[] = {0,0,0,0};
271       static double pErr[] = {0,0,0,0};
272       //guiUpdateThread->sendLog(intToString((int)segLeft));*/
273       while(segLeft<2)
274       {
275         if(phase<(dutyFactor-.5))
276         {
277           state=STANCE_1;
278         }
279         else if(phase<.5)
280         {
281           state=SWING_L;
282         }
283         else if(phase < dutyFactor)
284         {
285           state=STANCE_2;
286         }
287         else
288         {
289           state=SWING_R;
290         }
291
292
293         if(state == STANCE_1)
294         {
295           double factor = phase/(dutyFactor-.5);
296
297           double xPosL = half_backward*(1-factor)+backward*factor;
298           double yPosL = cartY;
299
300           double xPosR = forward*(1-factor)+half_forward*factor;
301           double yPosR = cartY;
302
303           double cartPositions[] = {xPosL,yPosL,xPosR,yPosR};
304
305           double a[4];
306                   CubicSpline(0,0,1,0,1,a);
307                   double t=factor;
308                   double rPerc=a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0];
309
310
311           FFLinkControl(cartPositions,motorPositions,rPerc);
312
313         }
```

```
314         else if(state == SWING_L)
315         {
316           double factor = (phase-(dutyFactor-.5))/(.5-(dutyFactor-.5));
317           filters[0]->clear();
318           filters[1]->clear();
319
320
321           double xPosL,yPosL;
322           double a[4];
323           double t;
324           if(factor<.5)
325           {
326             CubicSpline(0,0,liftHeight,0,.5,a);
327             t=factor;
328           }
329           else
330           {
331             CubicSpline(liftHeight,0,0,0,.5,a);
332             t=factor-.5;
333           }
334           yPosL=cartY+a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0];
335
336           if(factor<swing_time_1)
337           {
338             CubicSpline(backward,speed*(1-dutyFactor)*period,backward*gamma,0,
                  swing_time_1,a);
339             t=factor;
340           }
341           else if(factor<swing_time_2)
342           {
343             CubicSpline(backward*gamma,0,forward*gamma,0,swing_time_2-swing_time_1,a);
344             t=factor-swing_time_1;
345           }
346           else
347           {
348             CubicSpline(forward*gamma,0,forward,speed*(1-dutyFactor)*period,
                  swing_time_1,a);
349             t=factor-swing_time_2;
350           }
351           xPosL=a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0];
352
353
354           double xPosR = half_forward*(1-factor)+half_backward*factor;
355           double yPosR = cartY;
356           //double yPosR = -sqrt(stance_length*stance_length-xPosR*xPosR);
357           //double yPosL = sqrt(.2499 - xPosL * xPosL);
358
359           double cartPositions[] = {xPosL,yPosL,xPosR,yPosR};
360
361           FFLegControl(cartPositions,motorPositions);
362
363           double motorPositions2[4];
364           InvKin(cartPositions,jointPositions,motorPositions2);
365
366           motorPositions[0]=motorPositions2[0];
367           motorPositions[1]=motorPositions2[1];
368           correction[0]=0;
369           correction[1]=0;
370         }
371         else if(state == STANCE_2)
372         {
373           double factor = (phase-.5)/(dutyFactor-.5);
374
375           double xPosR = half_backward*(1-factor)+backward*factor;
376           double yPosR = cartY;
```

```
377
378         double xPosL = forward*(1-factor)+half_forward*factor;
379         double yPosL = cartY;
380
381         double cartPositions[] = {xPosL,yPosL,xPosR,yPosR};
382
383         double a[4];
384             CubicSpline(0,0,1,0,1,a);
385                 double t=factor;
386                 double rPerc=1-(a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0]);
387         FFLinkControl(cartPositions,motorPositions,rPerc);
388       }
389       else if (state == SWING_R)
390       {
391         filters[2]->clear();
392         filters[3]->clear();
393         double factor = (phase-dutyFactor)/(1-dutyFactor);
394
395         double xPosR,yPosR;
396         double a[4];
397         double t;
398         if(factor<.5)
399         {
400           CubicSpline(0,0,liftHeight,0,.5,a);
401           t=factor;
402         }
403         else
404         {
405           CubicSpline(liftHeight,0,0,0,.5,a);
406           t=factor-.5;
407         }
408         yPosR=cartY+a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0];
409
410         if(factor<swing_time_1)
411         {
412           CubicSpline(backward,speed*(1-dutyFactor)*period,backward*gamma,0,
                 swing_time_1,a);
413           t=factor;
414         }
415         else if(factor<swing_time_2)
416         {
417           CubicSpline(backward*gamma,0,forward*gamma,0,swing_time_2-swing_time_1,a);
418           t=factor-swing_time_1;
419         }
420         else
421         {
422           CubicSpline(forward*gamma,0,forward,speed*(1-dutyFactor)*period,
                 swing_time_1,a);
423           t=factor-swing_time_2;
424         }
425         xPosR=a[3]*pow(t,3)+a[2]*pow(t,2)+a[1]*t+a[0];
426
427         double xPosL = half_forward*(1-factor)+half_backward*factor;
428         //double yPosL = -sqrt(stance_length*stance_length-xPosL*xPosL);
429         double yPosL = cartY;
430         //double yPosL = sqrt(.2499 - xPosL * xPosL);
431
432         double cartPositions[] = {xPosL,yPosL,xPosR,yPosR};
433
434         FFLegControl(cartPositions,motorPositions);
435
436         double motorPositions2[4];
437         InvKin(cartPositions,jointPositions,motorPositions2);
438
439         motorPositions[2]=motorPositions2[2];
```

```
440           motorPositions[3]=motorPositions2[3];
441           correction[2]=0;
442           correction[3]=0;
443         }
444
445         double direction;
446         if(speed>0)
447           direction=1;
448         else
449           direction=-1;
450         static double prevCorrection[] = {0,0,0,0};
451
452
453
454         double correctionSpeedMax = 550000./1.*2.*PI/252000.;
455         for(int i=0;i<4;i++)
456         {
457
458           if(correction[i]!=0)
459           {
460             correction[i]-=Kd*linkVel[i];
461
462           }
463
464           double dcorrection = correction[i]-prevCorrection[i];
465           if(dcorrection/dt > correctionSpeedMax)
466           {
467             correction[i]=prevCorrection[i]+correctionSpeedMax*dt;
468           }
469           if(dcorrection/dt < -correctionSpeedMax)
470           {
471             correction[i]=prevCorrection[i]-correctionSpeedMax*dt;
472           }
473
474           motorPositions[i]+=correction[i];
475           prevCorrection[i]=correction[i];
476         }
477
478         phase+=direction*2*servoLoopTime/period;
479
480         while(phase>1)
481         {
482           phase--;
483           cycleNum++;
484         }
485         while(phase<0)
486           phase++;
487
488         int d[4];
489         int speedFrac=4;
490
491         for(int i=0;i<4;i++)
492         {
493
494           d[i]=round((motorPositions[i]-prevMot[i])*252000./(2.*PI));
495           if(d[i]> 550/speedFrac)
496             d[i]=550/speedFrac;
497
498           if(d[i]<-550/speedFrac)
499             d[i]=-550/speedFrac;
500
501
502           prevMot[i]+=d[i]*(2.*PI)/252000.;
503         }
504         galilCon->command("CD"+intToString(d[3])+","+intToString(d[2])+","
```

```
505                 +intToString(d[0])+","+intToString(d[1]));
506
507
508         segLeft++;
509       }
510     }
511
512
513
514
515
516
517     if(sendCommand)
518     {
519       if(interpVelocities)
520       {
521         for(int i=0;i<4;i++)
522         {
523           motorVelocities[i]=(motorPositions[i]-prevMot[i])/dt;
524         }
525       }
526
527       int p[4],v[4];
528       for(int i=0;i<4;i++)
529       {
530         v[i]=(int) abs(motorVelocities[i]*252000./(2.*PI));
531         p[i]=(int) (motorPositions[i]*252000./(2.*PI));
532       }
533
534       galilCon->command("SP"+intToString(v[3])+","+intToString(v[2])+","
535                 +intToString(v[0])+","+intToString(v[1]));
536
537       galilCon->command("PA"+intToString(p[3])+","+intToString(p[2])+","
538                 +intToString(p[0])+","+intToString(p[1])+";");
539
540       prevMot[0]=motorPositions[0];
541       prevMot[1]=motorPositions[1];
542       prevMot[2]=motorPositions[2];
543       prevMot[3]=motorPositions[3];
544     }
545     prevTime = currentTime;
546     if(cycleNum>=cycles)
547     {
548       running=false;
549       galilCon->command("CD0,0,0,0=0");
550
551       stringstream s1;
552       s1<< "Speed " << speed_out_prev;
553       guiUpdateThread->sendLog(s1.str());
554     }
555     //cout << "End State Machine" << endl;
556 }
557
558 void WalkingStateMachine::InvKin(double footPosition[], double linkPosition[],double
        motorPosition[])
559 {
560   double x[2];
561   double y[2];
562   x[0]=footPosition[0];
563   y[0]=footPosition[1];
564   x[1]=footPosition[2];
565   y[1]=footPosition[3];
566
567   double deg = 3.5*PI/180.;
568   const double LEG_SPREAD = BODY_WIDTH-LEG_WIDTH;
```

```
569   const double BOOM_R_LENGTH = BOOM_LENGTH+LEG_WIDTH/2;
570
571   double theta=-record->getReading(ROLL_ANGLE);
572     //theta=0;
573
574   x[0]=x[0]*((BOOM_R_LENGTH + LEG_SPREAD) * cos(theta) - (y[0]+LEG_SPREAD/2*sin(theta
          ))*tan(theta))/(BOOM_R_LENGTH+LEG_SPREAD/2)/cos(theta);
575   x[1]=x[1]*(BOOM_R_LENGTH * cos(theta) - (y[1]+LEG_SPREAD/2*sin(theta))*tan(theta))
          /(BOOM_R_LENGTH+LEG_SPREAD/2)/cos(theta);
576
577
578   y[0]=(y[0]-LEG_SPREAD/2*sin(theta))/cos(theta);
579   y[1]=(y[1]+LEG_SPREAD/2*sin(theta))/cos(theta);
580
581
582   double newx=cos(deg)*x[0]+sin(deg)*y[0];
583   double newy=-sin(deg)*x[0]+cos(deg)*y[0];
584   x[0]=newx;
585   y[0]=newy;
586
587   newx=cos(deg)*x[1]+sin(deg)*y[1];
588   newy=-sin(deg)*x[1]+cos(deg)*y[1];
589   x[1]=newx;
590   y[1]=newy;
591
592
593   double length_1 = sqrt(x[0]*x[0]+y[0]*y[0]);
594   double length_2 = sqrt(x[1]*x[1]+y[1]*y[1]);
595
596   cout << "L" << length_1 << endl;
597
598   if(length_1>.499)
599   {
600     x[0]*=.499/length_1;
601     y[0]*=.499/length_1;
602     }
603   if(length_2>.499)
604   {
605     x[1]*=.499/length_2;
606     y[1]*=.499/length_2;
607   }
608
609   double den = 2 * THIGH_LENGTH * THIGH_LENGTH;
610   double t2[2],t1[2],num,a,b;
611   for(int i = 0 ; i<2 ; i++)
612   {
613     num = pow(x[i],2) + pow(y[i],2) - 2*pow(THIGH_LENGTH,2);
614     t2[i] = abs(acos(num/den));
615     b = THIGH_LENGTH + THIGH_LENGTH * cos(t2[i]);
616     a = THIGH_LENGTH * sin(t2[i]);
617     t1[i] = atan2(b,a) - atan2(sqrt(pow(a,2)+pow(b,2)-pow(x[i],2)),x[i]);
618   }
619   linkPosition[0]=t1[0];
620   linkPosition[1]=t2[0];
621   linkPosition[2]=t1[1];
622   linkPosition[3]=t2[1];
623
624   motorPosition[0]=t1[0];
625   motorPosition[1]=t1[0]+t2[0];
626   cout << motorPosition[0] << endl;
627   cout << motorPosition[1] << endl;
628
629   motorPosition[2]=t1[1];
630   motorPosition[3]=t1[1]+t2[1];
631
```

```
632 }
633
634 void WalkingStateMachine::FFLinkControl(double cartPositions[],double motorPositions
        [],double rightPercent)
635 {
636   double jointPositions[4];
637   InvKin(cartPositions,jointPositions,motorPositions);
638
639   double fy[2];
640
641   double theta = record->getReading(ROLL_ANGLE);
642   //theta=0;
643     fy[0]=WEIGHT * g *(1-rightPercent)/cos(theta);
644     fy[1]=WEIGHT * g *rightPercent/cos(theta);
645
646   //fy[0] = 1.1 * g * abs(FFootPos[1][0] / (FFootPos[0][0]-FFootPos[1][0]));
647   //fy[1] = 1.1 * g * abs(FFootPos[0][0] / (FFootPos[0][0]-FFootPos[1][0]));
648
649
650   for(int x = 0;x<4;x++)
651   {
652     double angle;
653     if(x%2==0)
654     {
655       angle = linkAngle[x];
656       //angle=motorPositions[x];
657       if(motorPositions[x]<0)
658       {
659         //motorPositions[x]+=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy[x/2])/
              SPRING_CONSTANT);
660         correction[x]=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy[x/2])/
              SPRING_CONSTANT);
661       }
662       else
663       {
664         correction[x]=0;
665       }
666     }
667     else
668     {
669       angle = linkAngle[x]+linkAngle[x-1];
670       //angle=motorPositions[x];
671       if(motorPositions[x]>0)
672       {
673         //motorPositions[x]+=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy[x/2])/
              SPRING_CONSTANT);
674         correction[x]=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy[x/2])/
              SPRING_CONSTANT);
675       }
676       else
677       {
678         correction[x]=0;
679       }
680     }
681   }
682 }
683 void WalkingStateMachine::FFLegControl(double cartPositions[],double motorPositions
        [])
684 {
685   double jointPositions[4];
686   InvKin(cartPositions,jointPositions,motorPositions);
687   double theta = record->getReading(ROLL_ANGLE);
688   //theta=0;
689   double  fy = WEIGHT * g / cos(-theta);
690   for(int x = 0;x<4;x++)
```

```
691   {
692     double angle;
693     if(x%2==0)
694     {
695       angle= linkAngle[x];
696       //angle=motorPositions[x];
697       if(motorPositions[x]<0)
698       {
699         //motorPositions[x]+=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy)/
              SPRING_CONSTANT);
700         correction[x]=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy)/SPRING_CONSTANT)
              ;
701       }
702       else
703       {
704         correction[x]=0;
705       }
706     }
707     else
708     {
709       angle = linkAngle[x]+ linkAngle[x-1];
710       //angle=motorPositions[x];
711       if(motorPositions[x]>0)
712       {
713         //motorPositions[x]+=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy)/
              SPRING_CONSTANT);
714         correction[x]=filters[x]->input(LINK_LENGTH*(-sin(angle)*fy)/SPRING_CONSTANT)
              ;
715       }
716       else
717       {
718         correction[x]=0;
719       }
720     }
721   }
722 }
723 void WalkingStateMachine::CubicSpline(double xo,double xo_dot, double xf,double
      xf_dot,double T,double params[])
724 {
725   params[0]=xo;
726   params[1]=xo_dot;
727   params[2]=-(T*xf_dot + 2*xo_dot*T-3*xf+3*xo)/(T*T);
728   params[3]=(-2*xf+2*xo+xo_dot*T+xf_dot*T)/(T*T*T);
729 }
```

BIBLIOGRAPHY

[1] TDK-Lambda Americas Inc., *SWS-600L Users Manual*. San Diego, CA.

[2] Galil Motion Control, *DMC-40x0 User Manual*. Rocklin, California.

[3] Avago Technologies, *HEDL-5540 Datasheet*. San Jose, California.

[4] D. P. Krasny, *Evolving Dynamic Maneuvers in a Quadruped Robot*. PhD thesis, The Ohio State University, Department of Electrical and Computer Engineering, 2005.

[5] R. L. Tedrake, *Applied Optimal Control for Dynamically Stable Legged Locomotion*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[6] M. Raibert, *Legged Robots that Balance*. Cambridge, MA: MIT Press, 1986.

[7] D. Reinkensmeyer, D. Aoyagi, J. Emken, J. Galvez, W. Ichinose, G. Kerdanyan, S. Maneekobkunwon, K. Minakata, J. Nessler, W. Timoszyk, K. Vallence, R. Weber, J. Wynne, R. Roy, R. D. Leon, J. Bobrow, S. Harkema, V. Edgerton, "Tools for understanding and optimizing robotic gait training," *Journal of Rehab. Res. Dev.*, vol. 43, pp. 657–670, 2006.

[8] J. M. Remic III, "Prototype leg design for a quadruped robot application," Master's thesis, The Ohio State University, Department of Mechanical Engineering, 2005.

[9] B. T. Knox, "Evaluation of a prototype series-compliant hopping leg for biped robot applications." Undergraduate Honors Thesis, The Ohio State University, Department of Mechanical Engineering, 2007.

[10] S. Curran, "Real-time computer control of a high-performance, series-elastic, articulated jumping leg." Undergraduate Honors Thesis, The Ohio State University, Department of Electrical and Computer Engineering, Aug. 2005.

[11] P. Huang, "Communication protocol improvement for a real-time computer control system for a jumping leg." Master's Project, The Ohio State University, Department of Electrical and Computer Engineering, Aug. 2006.

[12] P. Birkmeyer, "Real-time control of cycling in a high-performance leg with series-elastic actuation." Undergraduate Honors Thesis, The Ohio State University, Department of Electrical and Computer Engineering, June 2007.

[13] S. Curran, B. T. Knox, J. P. Schmiedler, and D. E. Orin, "Design optimization of series elastic actuators for dynamic robots with articulated legs," in *Proc. of 2008 ASME Dynamic Systems and Control Conference*, vol. 1, no. 1, pp. 1–9, ASME, October 2008.

[14] Pratt, J. E., Chee, M. C., Torres, A., Dilworth, P., & Pratt, G. A., "Virtual model control: an intuitive approach for bipedal locomotion," *International Journal of Robotics Research*, vol. 20(2), pp. 129–143, 2001.

[15] J. W. Hurst and A. A. Rizzi, "Series compliance for an efficient running gait," *IEEE Robotics and Automation Magazine*, vol. 15, pp. 42–51, 2008.

[16] R. Niiyama, A. Nagakubo, and Y. Kuniyoshi, "Mowgli: A bipedal jumping and landing robot with an artificial musculoskeletal system," *IEEE International Conference on Robotics and Automation*, vol. 25, No. 4, pp. 2546–2551, April 2007.

[17] B. Knox, "Design of a robot capable of dynamic maneuvers," Master's thesis, The Ohio State University, Department of Mechanical Engineering, 2008.

[18] T. Yang, E. R. Westervelt, J. P. Schmiedeler, and R. A. Bockbrader, "Design and control of a planar bipedal robot ERNIE with parallel knee compliance," *Autonomous Robots*, vol. 25, No. 4, pp. 317–330, November 2008.

[19] L. Palmer and D. E. Orin, "Attitude control of a quadruped trot while turning," in *Proc. of the 2006 IEEE Intl. Conference on Intelligent Robots and Systems*, Beijing, China, pp. 5743–5749, October 2006.

[20] M. Hester, "Stable control of jumping for a biped robot," Master's thesis, The Ohio State University, Department of Mechanical Engineering, 2009.

[21] Maxon Precision Motors, Inc., *EC Powermax-30 Datasheet*. Fall River, MA.

[22] Advanced Motion Controls, *ZBDC12A8 Datasheet*. Camarillo, CA.

[23] Motion Designs, Inc., "Brushless motor commutation," *Design Trends*, pp. 1–6, 2008. Available: http://www.motion-designs.com/.

[24] Maxon Motor USA, *HEDL-5540 Datasheet*.

[25] CUI Inc., *NSO-S Datasheet*. Tualatin, OR.

[26] Quantum Devices, Inc., *QD145 Datasheet*. Barneveld, WI.

[27] Bourns, Inc., *6637 Precision Potentiometer Data Sheet*. Riverside, CA.

[28] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, March 1986.

[29] A. G. Feldman, "Superposition of motor programs i. rhythmic forearm movements in man," *Neuroscience*, vol. 5, pp. 81–90, 1980.

[30] F. A. Mussa-Ivaldi, S. F. Giszter, and E. Bizzi, "Linear combinations of primitives in vertebrate motor control," in *Proceedings of the National Academy of Science*, vol. 91, pp. 7534–7538, 1994.

[31] A. Antoniou, *Digital Filters: Analysis and Design*. McGraw-Hill, 1979.

[32] L. R. Palmer and D. E. Orin, "Qudarupedal running at high speed over uneven terrain," in *Proc. of the 2007 IEEE Intl. Conference on Intelligent Robots and Systems*, San Diego, CA, pp. 303–308, October 2007.

[33] S. Rodenbaugh, "*RobotBuilder* : a graphical software tool for the rapid development of robotic dynamic simulations," Master's thesis, Ohio State University, Department of Electrical & Computer Engineering, 2003.

[34] D. E. Orin, "Supervisory control of a multilegged robot," *International Journal of Robotics Research*, vol. 1, No. 1, pp. 79–91, 1982.

[35] A. Herum, "Personal communication." Galil Motion Controls, Rocklin, California.

[36] Wolfram Research, Inc. , "The Wolfram functions site." Online Resource, 2009. Available: http://functions.wolfram.com/.

[37] M. D. Lutovac, D. V. Tosic, "Elliptic rational functions," *The Mathematica Journal, Wolfram Media*, vol. 9, No.3, pp. 598–608, 2005.

[38] M. D. Lutovac, D. V. Tosic, B. L. Evans, *Filter Design for Signal Processing using MATLAB and Mathematica*. New Jersey, USA: Prentice Hall, 2001.

[39] J. J. Craig, *Introduction to Robotics Mechanics and Control*. Upper Saddle River, NJ: Prentice Hall, 3 ed., 2005.